STADIA
HELSINGIN AMMATTIKORKEAKOULU

TEKNIIKAN JA LIIKENTEEN TOIMIALA

Tietotekniikka

Ohjelmistotekniikka

INSINÖÖRITYÖ

PELIN KEHITYS WIN32-ALUSTALLA KÄYTTÄEN OPEN SOURCE -TYÖKALUJA

Työn tekijä: Juhana Hirvilahti
Työn valvoja: Miikka Mäki-Uuro
Työn ohjaaja: Miikka Mäki-Uuro

Työ hyväksytty: __. __. 2007

Miikka Mäki-Uuro
filosofian maisteri

STADIA
HELSINGIN AMMATTIKORKEAKOULU

ALKULAUSE

Ensimmäinen yritykseni tehdä opinnäytetyö oli nykyisen työnantajani toimeksiannosta. Tuolloin aiheena oli Midgard CMS -sovelluspalvelimen integrointi Jlibrary-dokumenttivarastoon. Midgardin Java-ohjelmointirajapinta ei kuitenkaan koskaan saavuttanut niin vakaata tilaa, että insinöörityön tekeminen olisi tuolloin ollut mahdollista.

Vastoinkäymisten jälkeen päätin ottaa opinnäytetyöni aiheeksi ohjelmoinnin osa-alueen, joka itseäni eniten kiinnostaa, peliohjelmoinnin. Lähetin muutamia kyselyitä alan yrityksiin, mutta mitään järkevää aihetta ei tuntunut löytyvän, joten ainoaksi vaihtoehdoksi jäi tehdä työ täysin itsenäisesti. Aluksi tarkoituksena oli ohjelmoida alusta asti kohtalaisen yksinkertainen pelimoottori ja peli, mutta tutkiskeltuani saatavilla olevia Open Source -pelimoottoreita ja -ohjelmakehyksiä päädyin tekemään tämän opinnäytetyön käyttäen OGRE 3D -ohjelmistokehystä.

Työn tekeminen ilman ammattilaisen opastusta on tietysti haastavaa ja pahimpien sudenkuoppien välttäminen on täysin omalla vastuulla. Toisaalta parhaimmillaan itsenäisen työn tekeminen on erittäin palkitsevaa ja opettavaista, joten uskon kehittyneeni valtavasti tämän insinöörityön aikana, niin ohjelmoijana, ihmisenä ja insinöörinäkin.

Erityiset kiitokset kuuluvat kaikille Open Source -ohjelmistojen kehittäjille, joita ilman tämän työn tekeminen ei olisi ollut mahdollista. Iso kiitos kuulu myös ohjelmistotekniikan opettajalleni Simo Silanderille, jonka opetus on ollut ensiluokkaista sekä työn valvojalle Miikka Mäki-Uurolle neuvoista ja opastuksesta.

Helsingissä 22.10.2007

Juhana Hirvilahti

STADIA
HELSINGIN AMMATTIKORKEAKOULU

INSINÖÖRITYÖN TIIVISTELMÄ

Tekijä: Juhana Hirvilahti

Työn nimi: Pelin kehitys win32-alustalla käyttäen Open Source -työkaluja

Päivämäärä: 22.10.2007 Sivumäärä: 49 s.

Koulutusohjelma: Tietotekniikka Suuntautumisvaihtoehto: Ohjelmistotekniikka

Työn valvoja: Miikka Mäki-Uuro

Työn ohjaaja: Miikka Mäki-Uuro

Peliteollisuus on nykyään erittäin suuri ohjelmistokehityksen ala, joten on ajankohtaista tutustua Open Source -työkalujen ja kirjastojen tarjoamiin mahdollisuuksiin. Visuaalisen viihteen tuottamiseen tarvitaan yleensä C++-ohjelmointitaidon lisäksi mallinnustaitoa ja kuvankäsittelytaitoa. Tämän lisäksi äänten tuottaminen on erittäin suuri osa toimivan kokonaisuuden saavuttamiseksi. Tässä työssä käsitellään kaikki osa-alueet ja tutkitaan Open Source -työkalujen soveltuvuutta pelin kehitykseen win32-alustalla. Lopputuloksena syntyy täysin pelattava, tosin yksinkertainen peli CrazyBunny.

Työn alussa esitellään kaikki käytettävät työkalut jotka kuuluvat tarvittavaan kehitysympäristöön. Tähän esittelyyn kuuluvat myös olennaisena osana työkalujen asennuksen läpikäynti sekä käyttöönotto. Työn perustana on käytetty OGRE-ohjelmistokehystä, joka ei ole varsinainen pelimoottori. Puuttuvia ominaisuuksia on lisätty käyttämällä CEGUI-kirjastoa käyttöliittymien tekoon sekä FMOD-kirjastoa äänijärjestelmän toteutukseen. Muita käytettyjä työkaluja ovat Code::Blocks-kehitysympäristö, Blender-mallinnusohjelma ja Audacity-äänieditori.

Pelisovelluksen toteutuksen pohjana on käytetty State-sunnittelumalliin perustuvaa järjestelmää pelitiloja hallintaan. Tässä mallissa pelin päävalikko, pelitila ja pelin loppu on erotettu omiksi tilaluokikseen, jolloin sovelluksesta saadaan helpommin hallittava. Päävalikossa tärkein osa on itse valikoiden toteutus CEGUI-kirjaston avulla. Pelitilan toteutuksessa tutustutaan OGRE:n visuaalisiin ominaisuuksiin kuten ympäristöön, valoihin, varjoihin, kuva-alustoihin ja visuaalisiin tehosteisiin. Tämän lisäksi peliin on toteutettu äänet suositulla FMOD-kirjastolla, jota useat isot alan yritykset käyttävät kaupallisissa tuotteissaan.

Avainsanat: pelimoottori, ohjelmistokehys, peliteollisuus, peli, C++, ohjelmointi

STADIA
HELSINGIN AMMATTIKORKEAKOULU

ABSTRACT

Name: Juhana Hirvilahti

Title: Developing a 3D Game on win32 Platform Using Open Source Tools

Date: 22 October 2007 Number of pages: 49

Department: Information Technology Study Programme: Software Engineering

Instructor: Miikka Mäki-Uuro

Supervisor: Miikka Mäki-Uuro

These days the game industry is a very large field of software development and therefore it is timely to familiarise with some of the open source tools and libraries. There are many aspects in producing visual entertainment and mastering C++-programming is not enough to meet the requirements. Image manipulation and 3D-modelling skills are also needed for reaching a satisfactory result. This thesis considers all of the basic requirements for game programming and analyses the possibilities of open source tools in game programming on win32 platform. As a result, a simple game called CrazyBunny is produced.

All of the tools and libraries used in this thesis are explained at the beginning with detailed instructions on how to install a development environment on the win32 platform. The OGRE rendering engine is used as a base for the game implementation. OGRE is not a game engine so some additional libraries are needed. The FMOD library is used for sounds and CEGUI library for creating menus. Additional tools like Blender modelling software, Audacity audio editor and Code::Blocks development environment are also used in the game making process.

The State design pattern is used as a base for the game implementation and it is used for controlling the different game states. In this system the game's main menu, actual game and the end of the game are implemented as separate states, which gives a lot of control over the program. The most important part of the main menu implementation is using the CEGUI library for creating a simple menu system. The implementation of the actual game state explains how OGRE can be used to create a scene using basic world geometry, lights, shadows, overlays and particle effects. A simple sound system is also created using the FMOD library, which is very popular in the industry.

Keywords: game engine, framework, game industry, game, C++, programming

SISÄLLYS

1 JOHDANTO

Peliteollisuus on viime vuosina kasvanut huimasti ja kukapa ei nykyaikana olisi pelannut tietokone- tai konsolipeliä puhumattakaan mobiilipeleistä. Peleistä on tullut monille jokapäiväistä ajanvietettä ja peliteollisuudessa käytetäänkin huimia budjetteja tämän digitaalisen viihteen kehitykseen ja markkinointiin. Tästä syystä onkin ajankohtaista tutustua tekniikoihin, ohjelmakirjastoihin ja työkaluihin, joita pelisovelluksissa käytetään.

Tämä työ on tutkimus siitä, kuinka nykyaikaisten kolmiulotteisten pelien ohjelmointi onnistuu saatavilla olevilla avoimen lähdekoodin työkaluilla. Kaikki työkalut ja kirjastot on tarkoin valittu sen mukaan, kuinka hyvin kukin projekti on dokumentoitu sekä yhteensopiva muiden käytettyjen työkalujen kanssa. Ohjelmointikielenä on käytetty C++:aa, joka on jo pitkään ollut suosituin ohjelmointikieli pelinkehittäjien keskuudessa.

Valitessaan avoimen lähdekoodin työkaluja ja kirjastoja on oltava tarkkana lisenssien kanssa. Tämä pätee lähinnä, mikäli aikoo julkaista ohjelmansa kaupallisessa tarkoituksessa. GPL (GNU General Public License) takaa sen, että tämän lisenssin alaisuudessa olevan ohjelman lähdekoodi jaetaan eteenpäin. Sama pätee, mikäli omassa ohjelmassaan käyttää GPL-lisenssin alaisuudessa olevaa ohjelmakirjastoa. LGPL (GNU Lesser General Public License) on kompromissi GPL:n ja muiden yksinkertaisempien lisenssien välillä. Tarkoitus on, että LGPL-lisenssin alaisia ohjelmakirjastoja voidaan käyttää kaupallisissa sovelluksissa ilman, että ohjelman lähdekoodia tarvitsee julkaista. Esimerkiksi GNU C-kirjasto on LGPL-lisensoitu.

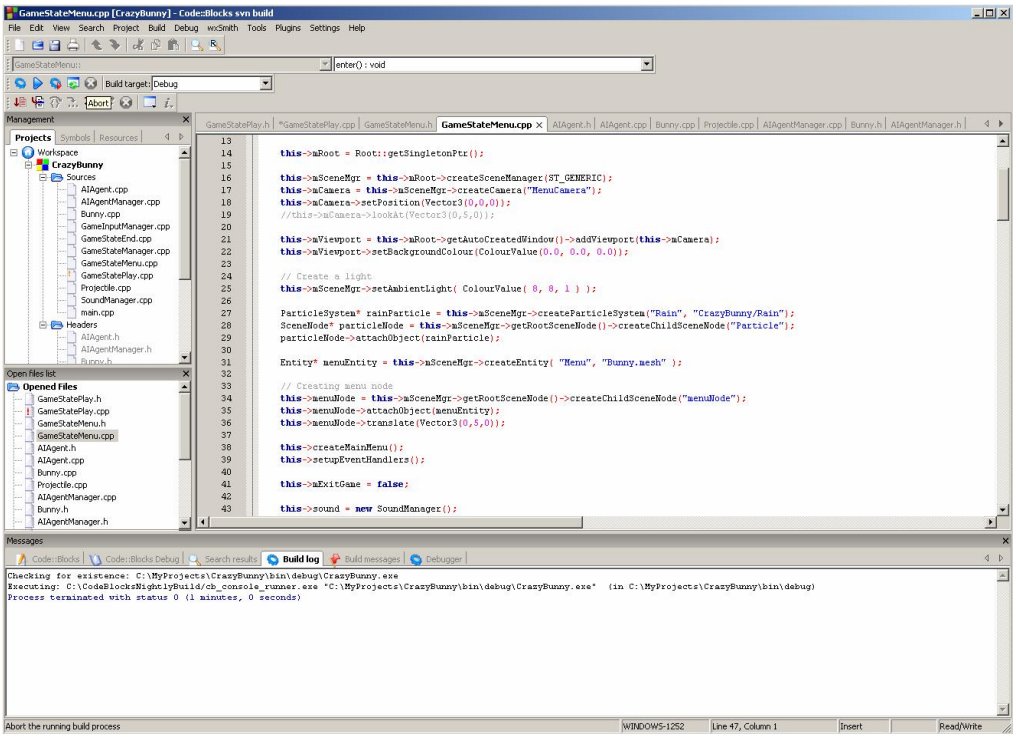
Insinöörityöni tarkoituksena on koota avoimen lähdekoodin työkaluista kehitysympäristö win32-alustalle ja käsitellä pelinkehitystä tässä ympäristössä. Aiheena pelinkehitys on todella laaja ja monipuolinen, joten tässä työssä keskitytään lähinnä aivan perusteisiin. Lopputuloksena syntyy täysin toimiva ja pelattava tietokonepeli.

2 KÄYTETTÄVÄT OPEN SOURCE –TYÖKALUT

Kehitysympäristö tätä työtä varten on koottu useista avoimen lähdekoodin työkalusta ja ohjelmakirjastoista. Ohjelmointiin käytetään Code::Blocks IDE -integroitua kehitysympäristöä, joka tukee OGRE 3D -ohjelmistokehystä. Pelin lähdekoodi käännetään GCC:llä (GNU Compiler Collection), joka win32-ympäristössä joudutaan ajamaan MinGW:n (Minimalist GNU for Windows) läpi. Pelin graafinen käyttöliittymä luodaan käyttäen CEGUI-ohjelmakirjastoa ja yksinkertainen äänijärjestelmä toteutetaan FMOD-ohjelmakirjastolla. Kolmiulotteisten mallien luomiseen käytetään Blender-mallinnustyökalua.

2.1 Code::Blocks IDE -integroitu kehitysympäristö

Code::Blocks on erittäin kevyt ohjelmistojen kehitysympäristö verrattuna Borlandin tai Microsoftin vastaaviin tuotteisiin. Se tarjoaa hyvin selkeän käyttöliittymän (kuva 1) sekä mahdolisuuden käyttää työpöytiä (workspace). Ainoat tuetut ohjelmointikielet ovat C ja C++. Tämä on kuitenkin pelien ohjelmoinnin kannalta täydellistä, sillä C++ on suosituin ohjelmointikeli peliteollisuudessa.



Kuva 1: Code::Blocks käyttöliittymä

Code::Blocks on lisensoitu GPL2 (General Public License version 2) alle, joten se on täysin vapaasti ladattavissa Internetistä. Kehitysympäristö tukee myös ohjelmalaajennuksia (plugin), joiden tekeminen itse on mahdollista erikseen ladattavien kehitystyökalujen avulla. On kuitenkin huomattava, että itse tehdyt ohjelmalaajennuksen menevät suoraan GPL-lisenssin alaiseksi.

Code::Blocks tukee montaa eri kääntäjää kuten GCC:tä (GNU Compiler Collection) ja MSVC++:tä (Microsoft Visual C++), joista GCC on tässä työssä käytetty kääntäjä. Jotta win32-ympäristössä voidaan käyttää GCC-kääntäjää, on se ajettava MinGW:n läpi. Hyvä puoli Code::Blocks ohjelmistossa on se, että asennuksen voi tehdä integroidun MinGW:n kanssa.

Yksi tärkeimmistä ominaisuuksista ohjelman kehittäjän kannalta on toimiva vianetsin (debugger). Code::Blocks käyttää hyödykseen GDB-vianetsintä (GNU Project debugger), joka on monien mielestä yksi parhaimmista. UNIX-alustalla GDB on ehdottomasti suosituin vianetsin.

Code::Blocks vaikuttaa olevan suosittu peliohjelmoijien sekä muiden grafiikkaohjelmoijien parissa. Käyttöliittymästä löytyykin valmiita projektipohjia useille eri pelimoottoreille ja ohjelmistokehyksille kuten OGRE (http://www.ogre3d.org), Crystal Space (http://www.crystalspace3d.org), Irrlight (http://irrlicht.sourceforge.net) ja Lightfeather 3D (http://lightfeather.de).

2.2 MinGW-minimalistinen GNU Windows-ympäristöön

MinGW (”Minimalistic GNU for Windows”) on kokoelma vapaasti saatavilla olevista Windows-käyttöjärjestelmän otsikkotiedostoista ja kirjastoista, jotka on yhdistetty tavallisiin GNU-työkaluihin. Tämän ansiosta on mahdollista käyttää GCC-kääntäjää win32-alustalla natiivien Windows-ohjelmien kääntämiseen ilman, että ohjelmat ovat riippuvaisia kolmannen osapuolen dynaamisista kirjastoista. MinGW tarjoaa C-, C++- ja Fortran-kääntäjät sekä muita hyödyllisiä työkaluja.

MinGW käyttää hyödykseen Microsoftin Windows-käyttöjärjestelmän tarjoamia ohjelmakirjastoja joita ei ole lisensoitu GPL-lisenssin alaiseksi. Tästä syystä MinGW:n GCC-kääntäjällä käännettyjen ohjelmien lähdekoodia ei tarvitse luovuttaa yleiseen käyttöön ellei ohjelmassa käytä jotain muuta GPL-lisenssin alaista kirjastoa. Tämä poikkeaa muista saatavilla olevista Win-

dows-käyttöjärjestelmälle tarkoitetuista GCC-kääntäjistä, joissa kirjastot ovat GPL-lisenssin alaisia.

2.3 OGRE 3D SDK -ohjelmistokehys

OGRE (Object-Oriented Graphics Rendering Engine) on joustava 3D-moottori, joka on suunniteltu helpottamaan kolmiulotteisuutta hyödyntävien ohjelmien rakentamista. Luokkakirjasto kätkee taakseen kaikki ominaisuudet joita alemman tason grafiikkakirjastoilla (Direct3D ja OpenGL) on. Samalla se tarjoaa rajapinnan kolmiulotteisen maailman käsittelyyn.

Ohjelmistokehyksen lähdekoodi on lisensoitu LGPL-lisenssin (GNU Lesser General Public License) alaiseksi. Tämä käytännössä tarkoittaa sitä, että ohjelmistokehyksen lähdekoodiin tehdyt muutokset on julkaistava yleiseen käyttöön, mutta ohjelmistokehystä käyttävien ohjelmien lähdekoodia ei tarvitse julkaista. Tämä on kuitenkin suositeltavaa.

OGRE on suunniteltu siten, että sitä voidaan käyttää kaikenlaisten sovellusten rakentamiseen. Siksi se keskittyykin vain grafiikan tuottamiseen. Omaa peliä luodessa on kirjoitettava itse toteutus äänelle, tekoälylle, törmäysten tarkistukselle ja fysiikalle tai käytettävä muita kirjastoja näiden toteuttamiseen. Syy tähän on se, että erityyppiset pelit, simulaatiot tai muut ohjelmat tarvitsevat toisistaan poikkeavat 3D-moottorit toimiakseen. OGRE tarjoaakin ratkaisun, jota voidaan käyttää grafiikan hallinnointiin näissä kaikissa. Ohjelmakehys paisuisi aivan liian isoksi ja raskaaksi, jos kaikkien pelityyppien tai muiden ohjelmien vaatimukset otettaisiin huomioon.

Ideana on se, että ohjelmoija voi OGRE:n lisäksi valita haluamansa kirjastot ja käyttää näitä esimerkiksi oman pelimoottorinsa rakentamiseen ja lisätä vain ne ominaisuudet, joita tarvitaan.

Saatavilla on useita 3D-moottoreita, jotka ovat teknisesti kehittyneitä ja joilla on huimasti ominaisuuksia, mutta niiden dokumentointi on puutteellista eikä kunnon esimerkkejä löydy. OGRE-projekti tähtää enemmän hyvään suunnitteluun ja dokumentointiin, jotta ohjelmistokehystä voitaisiin käyttää tehokkaasti.

Kaikki uudet ominaisuudet, joita OGRE-moottoriin tehdään, käyvät läpi perusteellisen suunnittelun ja dokumentoinnin sillä laatua on hankala lisätä jäl-

kikäteen. Suunnittelumallien runsas käyttö ja kehitystiimin pitäminen pienenä takaavat OGRE:n kehittäjien mielestä hyvän sekä järjestelmällisen lopputuloksen.

2.4 FMOD-audiokirjasto

FMOD on nimenomaan pelejä ja muita multimediaohjelmia varten kehitetty nykyaikainen kolmiulotteinen äänijärjestelmä. Tarjolla on C sekä C++-ohjelmointirajapinnat, joista tässä työssä käytetään C-rajapintaa, sillä C++-rajapintaa ei voi valitettavasti käyttää MinGW:n kanssa.

FMOD-audiokirjasto on ainoa tässä työssä käytetty työkalu, joka ei varsinaisesti ole avointa lähdekoodia. Käyttäminen on kuitenkin täysin ilmaista ei-kaupallisissa projekteissa. Mikäli ohjelmansa julkaisee kaupalliseen levitykseen, on maksettava tiettyjen ehtojen mukaisia lisenssimaksuja. Lisenssiehdoista ja lisenssien hankkimisesta on saatavilla lisää tietoa verkkosivuilla osoitteessa (http://www.fmod.org/index.php/sales).

2.5 CEGUI-käyttöliittymäkirjasto

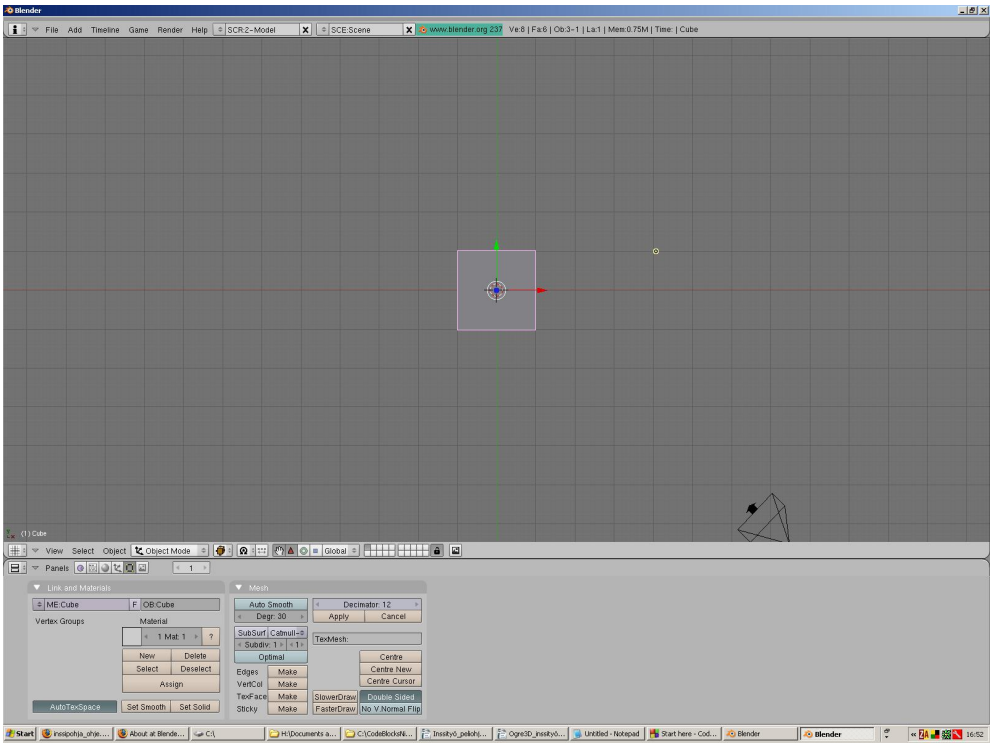
CEGUI (Crazy Eddie's GUI) on graafisten käyttöliittymien ohjelmointiin erikoistunut kirjasto joka tukee C++-ohjelmointikieltä. Se sopii käytännössä kaiken tyyppisten graafisten käyttöliittymien rakentamiseen jotka koostuvat ikkunoista ja erillisistä käyttöliittymäelementeistä (widget). Varsinainen joustavuus tulee ilmi siinä, että oman käyttöliittymänsä ulkoasun voi vapaasti muokata XML-kielellä tehtävillä tiedostoilla, joihin määritellään yksityiskohtaisesti kaikkien käyttöliittymäelementtien ulkonäkö. Omaa sovellustaan ei siis tarvitse uudelleen kääntää käyttöliittymän ulkonäköä muokatessa.

CEGUI-käyttöliittymäkirjasto on lisensoitu LGPL-lisenssin (GNU Lesser General Public License) alle, joten oman sovelluksen lähdekoodia ei tarvitse luovuttaa yleiseen käyttöön.

2.6 Blender 3D -mallinnusohjelmisto

Blender on 3D-mallinnukseen, animointiin sekä hahmonnukseen (render) soveltuva ohjelmisto, joka on saatavilla kaikille yleisimmille käyttöjärjestelmille. Blender on myös lisensoitu GPL-lisenssin alle ja näin ollen ilmaiseksi ladattavissa.

Blender on todella poikkeuksellinen ja monimutkaisen näköinen 3D-mallinnusohjelma verrattuna muihin markkinoilla oleviin tuotteisiin. Harjoittelun jälkeen kuitenkin huomaa, kuinka tehokas Blender todellisuudessa on. Näppäinkomennot on opeteltava ulkoa, sillä ne ovat aivan keskeinen asia Blenderillä mallinnettaessa. Peruskäyttöliittymä (kuva 2) on kohtuullisen yksinkertaisen näköinen, mutta se kätkee taakseen valtavan määrän ominaisuuksia ja on muokattavissa omien tottumusten mukaiseksi.



Kuva 2: Peruskäyttöliittymä

3 TYÖKALUJEN ASENTAMINEN WIN32-ALUSTALLE

3.1 MinGW-ympäristön asennus

Osoitteesta (http://www.mingw.org/download.shtml) imuroidaan tiedosto MinGW-5.0.2.exe ja ajetaan asennusohjelma. Asennus kannattaa tehdä esimerkiksi kansioon C:/MinGW.

Seuraavaksi imuroidaan alla listatut tiedostot ja puretaan ne listausjärjestyksessä kansioon C:/MinGW. Tämän jälkeen voidaan siirtyä Code::Blocks-kehitysympäristön asennukseen.

- gcc core 3.4.5
- g++ 3.4.5
- win32api 3.6
- binutils 2.17.50
- mingw runtime 3.10
- gdb 6.3.2

3.2 Code::Blocks-kehitysympäristön asennus

OGRE ei tällä hetkellä toimi vakaan Code::Blocks 1.0 RC2 -version kanssa, joten on asennettava joka yö käännettävä (Nightly build) versio. Tässä vaiheessa on myös tärkeää, että MinGW on asennettuna, jotta Code::blocks havaitsee tämän heti ensimmäisellä käynnistyskerralla.

3.2.1 Asennus

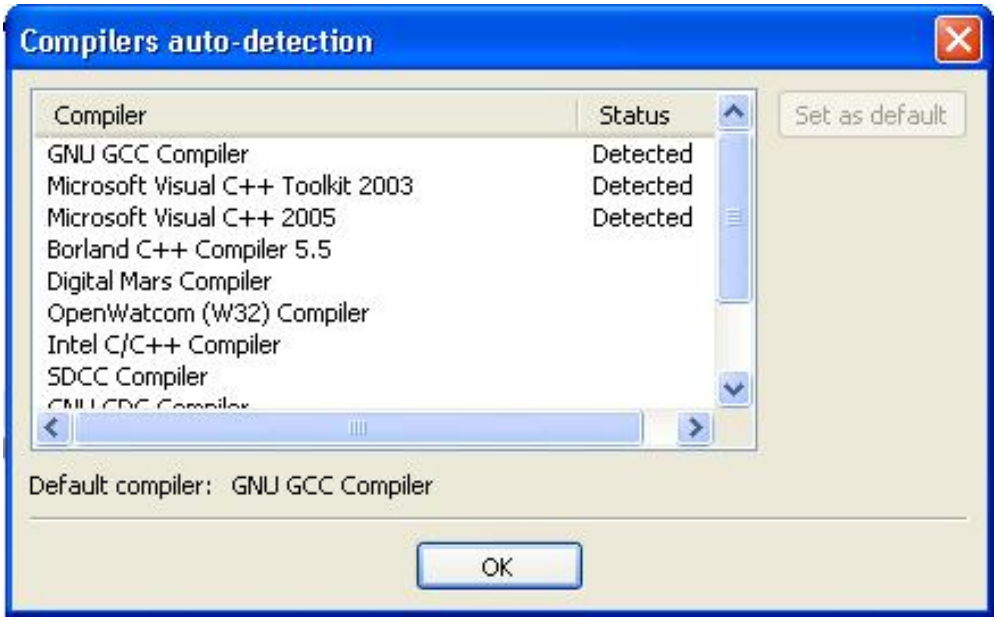
Viimeisin versio kehitysympäristöstä löytyy Code::Blocks-projektin foorumeilta (http://forums.codeblocks.org/index.php/board,20.0.html), josta käytännössä on ladattava kaksi tiedostoa, jotka on seuraavassa listattu.

- wxmsw26u_gcc_cb.7z
- CB_2006xxxx_revxxxx_win32.7z

Tiedostojärjestelmään kannattaa luoda esimerkiksi kansio C:/CodeBlocks, jonne kaksi yllä mainittua tiedostoa puretaan. Tämän lisäksi kansiosta C:/MinGW/bin on kopioitava tiedosto ”mingwm10.dll” kansioon C:/CodeBlocks/bin.

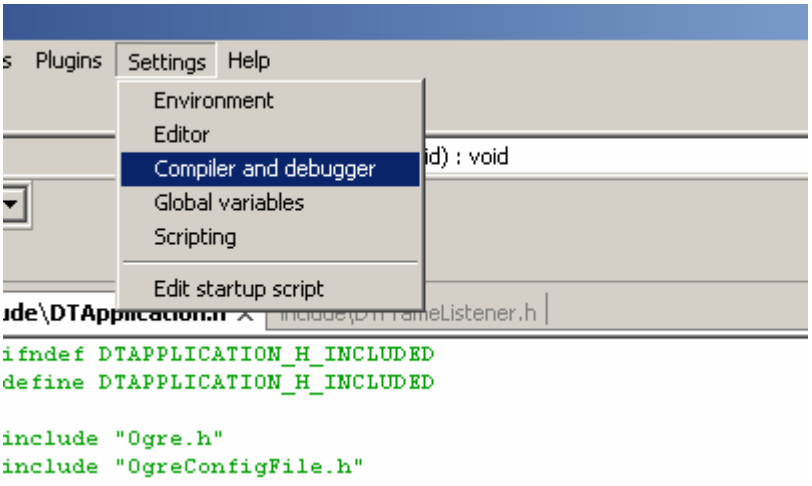
3.2.2 Asetukset

Ensimmäisellä kerralla, kun Code::Blocks käynnistetään, pitäisi näkyviin tulla ikkuna (kuva 3), josta voidaan asennetuista kääntäjistä valita mieleinen. Tässä tapauksessa valitaan ”GNU GCC Compiler” ja klikataan ”Set as default”.

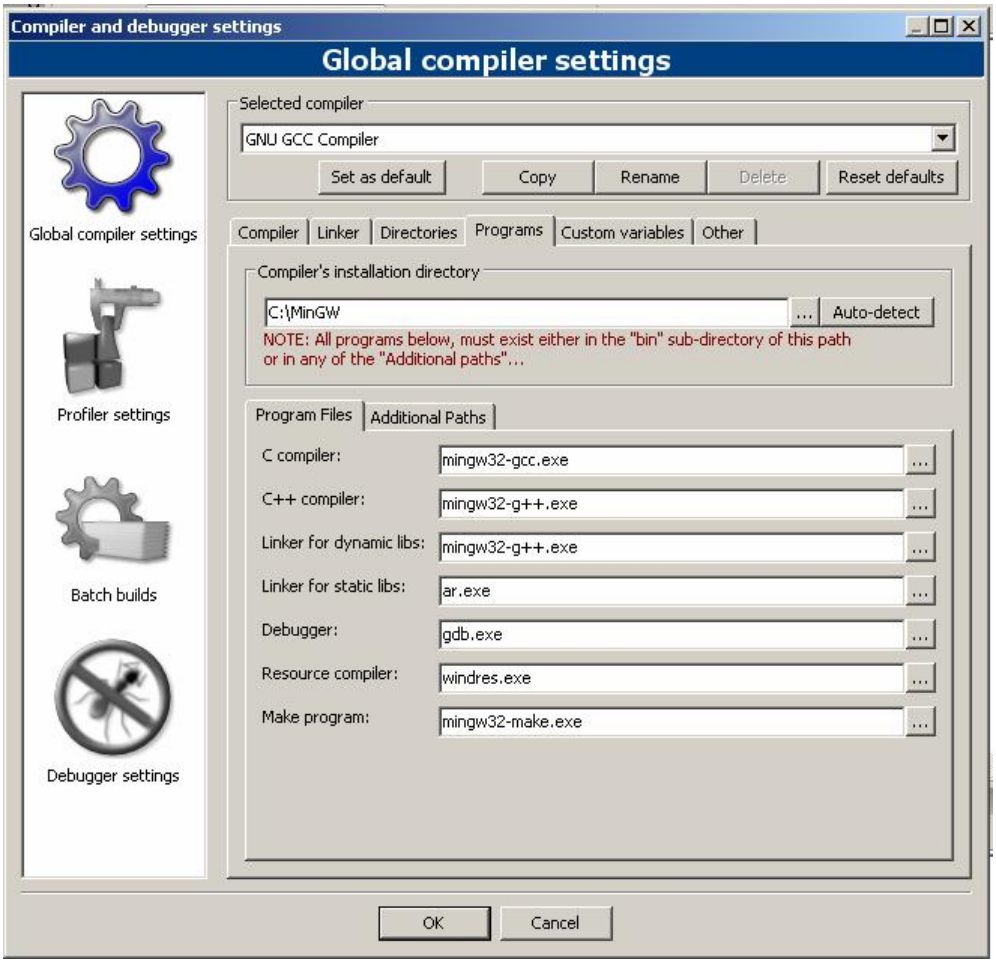


Kuva 3: Kääntäjän valinta

Seuraavaksi kannattaa avata kääntäjän asetukset (kuva 4) ja tarkistaa, että ”Programs”-välilehti on kuvan 5 mukainen. Code::Blocks on onnistuneesti asennettu mikäli näin on.



Kuva 4: Kääntäjä ja vianetsin (Compiler and Debugger)



Kuva 5: Ohjelmat (Programs)

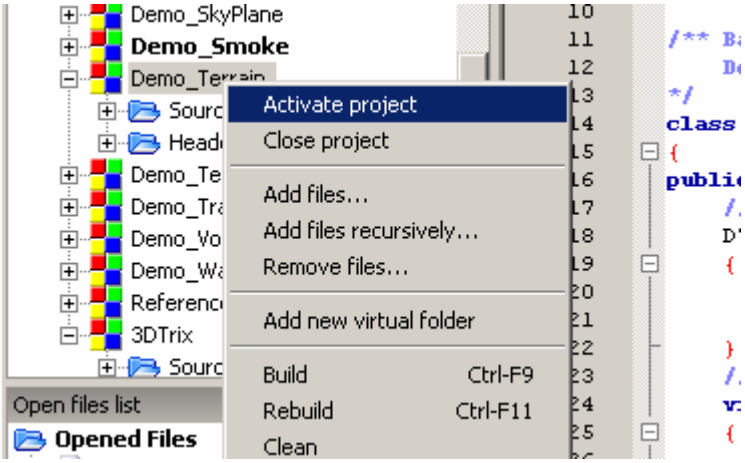
3.3 OGRE 3D -ohjelmistokehyksen asennus

Sivustolta (http://www.ogre3d.org/) löytyy linkin ”Downloads” alta kohta ”Current stable” josta imuroidaan uusin OGRE:n versio Code::Blocks-kehitysympäristölle. Tällä hetkellä uusin versio on OGRE 1.4.4 SDK for Code::Blocks + MinGW C++ Toolbox.

Seuraavaksi ajetaan asennus käynnistämällä tiedosto OgreSDKSetup1.4.4_CBMingW.exe. Mikäli asetuksia ei muuta, asentuu OGRE hakemistoon C:/OgreSDK.

Tässä vaiheessa kannattaa tarkistaa, että kaikki toimii kuten pitääkin. Helpoin tapa on avata Code::Blocks:lla tiedosto ”Samples.workspace” kansiosta C:/OgreSDK/Samples ja valita listalta esimerkiksi ”Demo_terrain”. Tämän kohdalla painetaan hiiren oikeaa näppäintä ja valitaan listalta ”Activate pro-

ject” (kuva 6). Seuraavaksi samasta valikosta valitaan ”Build”, joka kääntää ohjelman. Jos käännös menee läpi, on ympäristö oikein asennettu ja toimintakunnossa. Käännetyn ohjelman voi tämän jälkeen ajaa valitsemalla ”Run”.



Kuva 6: Projektin aktivointi

3.4 Blender-mallinnusohjelman asennus

Blender ei riipu mitenkään yllä asennetuista ohjelmista ja on siten täysin itsenäinen kokonaisuus. Asennus tapahtuu helpoiten imuroimalla Blenderin verkkosivulta (http://www.blender.org/download/get-blender/) Windows-käyttöjärjestelmälle tarkoitettu asennuspaketti ja ajamalla asennus.

3.5 Muut tarpeelliset työkalut

Erittäin tärkeä työkalu on jokin kuvankäsittelyohjelma tekstuurien tekoon ja muokkaukseen. Kuvien käsittelyyn käy mikä tahansa markkinoilla oleva ohjelma mutta tässä työssä käytetään GIMP-kuvankäsittelyohjelmaa (http://www.gimp.org/), joka on myös vapaaseen lähdekoodiin perustuva ohjelma.

Blender ei suoraan tue OGRE 3D -ohjelmistokehykstä, joten mallien vientiin tarvitaan ohjelmalaajennus nimeltä BlenderExporter, joka on vapaasti ladattavissa verkkosivunsivun (http://www.ogre3d.org/) tools-osiosta.

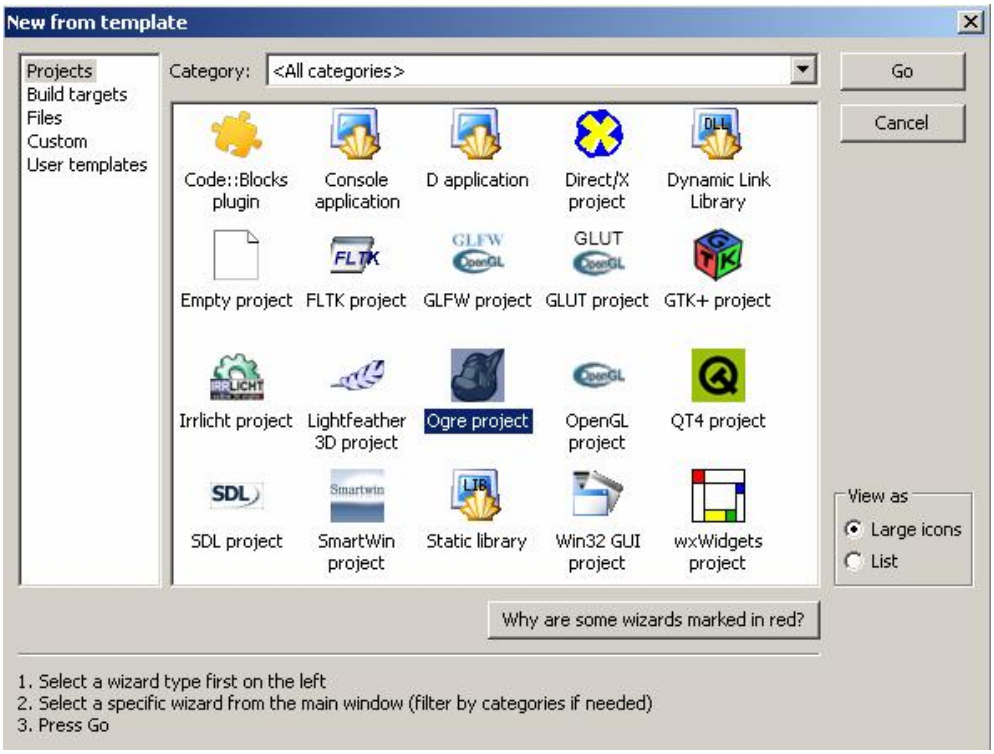
Pelin äänitehosteiden ja musiikin tekemiseen sekä muokkaamiseen tarvitaan äänieditori. Tätä tarkoitusta varten on valittu suhteellisen yksinkertainen Au-

dacity-äänieditorin (http://audacity.sourceforge.net/), joka on avoimen lähdekoodin ohjelma äänittämiseen ja äänten muokkaamiseen.

4 CODE::BLOCKS-PROJEKTIN LUONTI JA ASETUKSET

4.1 Uuden projektin luonti

Uusi projekti luodaan käyttäen Ogre-projektipohjaa siten, että valitaan ”File->New->Project”. Tämä avaa projektipohjavalikon (kuva 7), josta valitaan ”Ogre project”.



Kuva 7: Projektipohjan valinta

Seuraavalla sivulla projektille syötetään nimi ja hakemisto jonne projekti halutaan luoda. Loput ohjelman ehdottamat valinnat voi suoraan ohittaa sille ne ovat valmiiksi oikein. Lopuksi painetaan ”Finish”.

4.2 Projektin kääntäjän asetukset (Project build options)

Jotta projektin kääntäminen toimisi oikein, on tehtävä seuraavassa listauksessa esitetyt koko projektin laajuiset asetukset. Projektin asetuksiin pääsee painamalla projektin nimen päällä hiiren oikeaa näppäintä ja valitsemalla valikosta ”Build options”.

Compiler-osion Other options -välilehdelle tehdään alla olevan listan mukaiset asetukset.

- -mthreads
- -fmessage-length=0
- -fexception
- -fident

Linker-osion Other linker options -välilehdelle tehdään alla olevan listan mukaiset asetuket.

- -Wl,--enable-runtime-pseudo-reloc
- -Wl,--enable-auto-image-base
- -Wl,--add-stdcall-alias

Compiler-välilehdelle asetetaan hakemistopolku OGRE:n include-kansioon.

- $(OGRE_HOME)\include

Linker-välilehdelle asetetaan hakemistopolku OGRE:n bin-kansioon jossa $(TARGET_NAME) viittaa käytettävään käännöskohteeseen (debug tai release).

- $(OGRE_HOME)\bin\$(TARGET_NAME)

4.3 Kehitysversion kääntäjän asetukset (Debug build options)

Tässä on tärkeätä huomata, että linkkerille tulee antaa hakemistopolku tiedostoihin joiden nimi on muotoa ”nimi_d”. Tämä tarkoittaa sitä, että käytetään debug-versiota kyseisestä kirjastosta

Compiler-osion #defines-välilehdelle lisätään seuraavat asetukset.

- WIN32
- _DEBUG
- _WINDOWS

Linker-osion Link libraries -välilehdelle lisätään OGRE:n dynaaminen kirjasto.

- OgreMain_d

4.4 Julkaisuversion kääntäjän asetukset (Release build options)

Julkaisuversiossa linkkerille annetaan hakemistopolku varsinaiseen kirjastoon jota halutaan käyttää, eikä kyseisen kirjaston debug-versioon kuten edellisessä kohdassa.

Compiler-osion #defines-välilehdelle lisätään seuraavat asetukset.

- WIN32
- NDEBUG
- _WINDOWS

Linker-osion Link libraries -välilehdelle lisätään OGRE:n dynaaminen kirjasto.

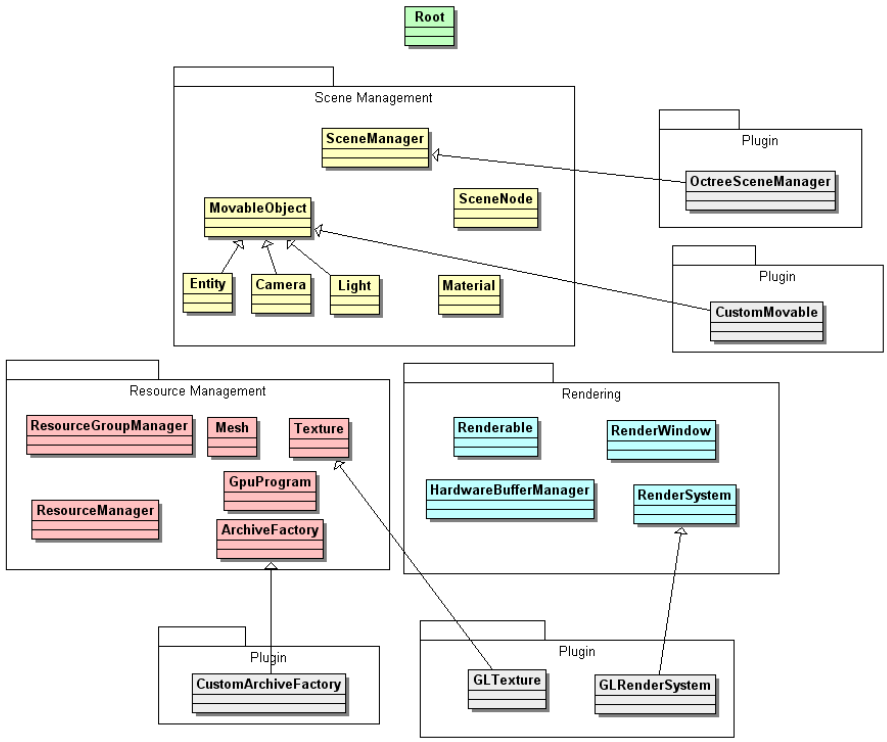
- OgreMain

5 OGRE 3D -OHJELMISTOKEHYKSEN TOIMINTA

OGRE on ohjelmoitu C++-ohjelmointikielellä, ja se käyttää tehokkaasti hyväkseen olio-ohjelmoinnille ominaisia piirteitä. Kaikki luokat ovat huolellisesti nimiavaruuksien alla, mikä ehkäisee turhat ongelmat nimien kanssa, kun OGRE on osana jotain muuta järjestelmää. Kapselointi lisää lähdekoodiin rakenteellisuutta ja samalla ohjelmakoodista tulee selkeämpää. Tämän li-

säksi OGRE käyttää runsaasti rajapintoja, joita vasten voi omat toteutuksensa ohjelmoida.

Seuraavassa UML-kaaviossa (Kuva 8) on esitettynä kaikki tärkeimmät luokat, joista OGRE koostuu. Kokonaisuuden voi kuitenkin ajatella jakautuvan neljään osaan, jotka on selitetty seuraavassa.



Kuva 8: UML-kaavio tärkeimmistä luokista

5.1 Juuritaso (Root)

Juuritaso (Root) on luokka, jonka ilmentymää voi sanoa pääsyreitiksi koko OGRE-järjestelmään. Kaikki päätason oliot kuten SceneManager, ResourceManager ja RenderSystem luodaan järjestelmän juuren kautta. Juuren tehtävä on enimmäkseen hallinnoida muita olioita, jotka varsinaisesti tekevät kaiken työn.

Juurioliota käytetään myös tekemään asetuksia järjestelmään. Tästä esimerkkinä voisi olla hahmonnusjärjestelmän, resoluution ja käytettävien värien määrän valinta.

Tärkein funktio on kuitenkin startRendering, joka aloittaa peleille ominaisen jatkuvan silmukan, jossa näyttö päivitetään jokaisella ohjelman kierroksella. Silmukka päättyy vain, jos kaikki ikkunat suljetaan tai jos jokin ohjelman kuuntelijoista (FrameListener) lopettaa silmukan.

5.2 Peliympäristön hallinta (Scene Management)

Peliympäristön hallintaan kuuluu ympäristösolmujen (SceneNode) paikkatietojen sekä varsinaisen sisällön pitäminen järjestyksessä. Tämä on toteutettu puumaisena rakenteena, jossa juurisolmuun voidaan liittää yksi tai useampia lapsisolmuja, jotka voivat myös sisältää omia lapsisolmuja. Tiettyyn lapsisolmuun liitetyt solmut voidaan esimerkiksi asettaa seuraamaan vanhempaansa, mikäli näin halutaan. Tämä on hyödyllistä monessa tilanteessa, sillä lapsisolmujen paikkatiedot päivittyvät automaattisesti vanhempansa paikkatietojen mukaan.

Ympäristösolmuihin on liitettävä ilmentymä (Entity), jos solmun halutaan näkyvän ruudulla. Ilmentymä on käytännössä jokin kolmiulotteinen malli. Tämä ei kuitenkaan ole pakollista, sillä joissakin tapauksissa on hyödyllistä ylläpitää näkymättömiä solmuja tiettyjä tehtäviä varten.

Peliympäristöä ei kuitenkaan voi hahmontaa (render) ruudulle ilman kameraa jonka tehtävä on kertoa järjestelmälle, mistä paikasta ja suunnasta ympäristö halutaan näyttää. Kamera onkin erikoistapaus tavallisesta ympäristösolmusta (SceneNode) mutta sitä voidaan käyttää samalla tavalla kuin tavallista ympäristösolmua. Erityisen hyödyllinen on ominaisuus, jolla kamera voidaan liittää tavalliseen solmuun samoin kuin tavallisiin solmuihin voidaan liittää lapsisolmuja.

Käytännössä kaikki peliympäristöön liittyvät operaatiot tehdään siis SceneManager-luokan ilmentymällä, joka on eniten käytetty luokka koko OGRE-järjestelmässä. SceneManeger-olio myös lähettää koko peliympäristön hahmonnusjärjestelmälle (RenderSystem), jossa varsinaiset kuvat muodostetaan ruudulle.

5.3 Resurssien hallinta (Resource Management)

Resurssien hallinta on tärkeä osa järjestelmää, sillä tekstuurien ja mallien lataaminen sekä käyttäminen järjestelmässä ovat keskeinen osa peliohjel-

mointia. Jokaista resurssityyppiä varten on olemassa oma Resource-Manager-luokka kuten TextureManager-luokka, jota käytetään tekstuurien lataamiseen. Nämä luokat pitävät huolen siitä, että tietty resurssi ladataan vain kerran, jonka jälkeen se on saatavilla koko järjestelmässä.

5.4 Hahmonnus (Rendering)

Hahmonnus tarkoittaa peliohjelmoinnissa vaihetta, jossa ruudulle tuotetaan kuva tietystä pelin tilanteesta. Tämä ruudulle piirrettävä grafiikka koostuu käytännössä ympäristösolmuihin (SceneNode) liitettävistä ilmentymistä (Entity), jotka ovat kolmiulotteisia malleja, tehosteita tai käyttöliittymäkomponentteja.

Tärkein luokka OGRE:n hahmonnuksessa on RendererSystem, joka on abstrakti luokka. Tämä määrittää yhteisen rajapinnan kaikille mahdollisille grafiikkakirjastoille kuten OpenGL ja Direct3D, jotka molemmat ovat OGRE:ssa tuettuna. RendererSystem-oliota ei kuitenkaan yleensä käytetä ohjelmassa suoraan, sillä yleensä kaikki kutsut ja asetukset tehdään peliympäristön hallinnan (ks. kpl 5.2) kautta. Käytettävän grafiikkakirjaston voi valita ogre.cnf-tiedostossa, jossa on mahdollista tehdä myös muita määrityksiä kuten käytettävä resoluutio ja värien määrä.

Olennainen osa hahmonnusta (rendering) on myös kamera, jota käytetään määrittämään se piste ja suunta, josta pelin ympäristö halutaan näyttää käyttäjälle. Kamera toimii normaalin ympäristösolmun tavoin, joten kameraan voi liittää esimerkiksi ilmentymän (Entity) tai kamera voidaan liittää toiseen ympäristösolmuun (SceneNode).

5.5 Helpoin tapa aloittaa pelin ohjelmointi

Helpoin tapa, jota tässä työssä ei kuitenkaan käytetä, aloittaa ohjelmointi OGRE-ohjelmistokehystä käyttäen on periyttää omalle luokalleen OGRE:n ExampleApplication-luokka (ExampleApplication.h). ExampleApplication-luokalla on aidosti virtuaalinen createScene()-funktio, jolle on kirjoitettava toteutus omassa ohjelmassaan. Kyseisessä funktiossa voi halutessaan luoda koko peliympäristön.

ExampleApplication-luokassa tehdään automaattisesti kaikki välttämättömät vaiheet OGRE:n käynnistyksessä. Tämä kuitenkin tarkoittaa sitä, että oma

ohjelma käyttää valmista kehyskuuntelijaa (FrameListener). Ohjelmaan kannattaa luoda OmaFrameListener-luokka, jolle periytetään ExampleFrameListener-luokka (ExampleFrameListener.h). Omassa kehyskuuntelijassa on tämän jälkeen ylikirjoitettava frameStarted()-funktio. Tässä funktiossa tehdään kaikki jokaisella ohjelman kierroksella tarvittavat operaatiot kuten näppäimistön syötteiden luku ja kameran paikan päivitys. ExampleApplication-luokalla on go()-funktio, joka tulee suorittaa pääohjelmassa. Mitään muuta ei pääohjelmassa tarvitse tehdä.

Tässä työssä ei kuitenkaan käytetä pohjana ExampleApplication-luokkaa, sillä se ei kunnolla pysty vastaamaan pelin tarpeita. ExampleApplication-luokka on vain esimerkki ja helppo tapa aloittaa tutustuminen OGRE-ohjelmistokehykseen. Pelisovelluksen pohjan voi todellisuudessa tehdä huomattavasti järkevämmin (ks. kpl. 6).

6 CRAZYBUNNY-PELIN KEHITYS

Tässä luvussa käydään hieman pintaa syvemmältä läpi ohjelman kehitystä käyttäen OGRE-ohjelmakehystä. OGRE ei kuitenkaan ole varsinainen pelimoottori, joten graafisen käyttöliittymän luontiin käytetään CEGUI-kirjastoa (ks. kpl. 6.3) ja äänijärjestelmä toteutetaan käyttäen FMOD-kirjastoa (ks. kpl. 6.11). Tuloksena syntyy yksinkertainen täysin pelattava peli, joka kantaa työnimeä CrazyBunny.

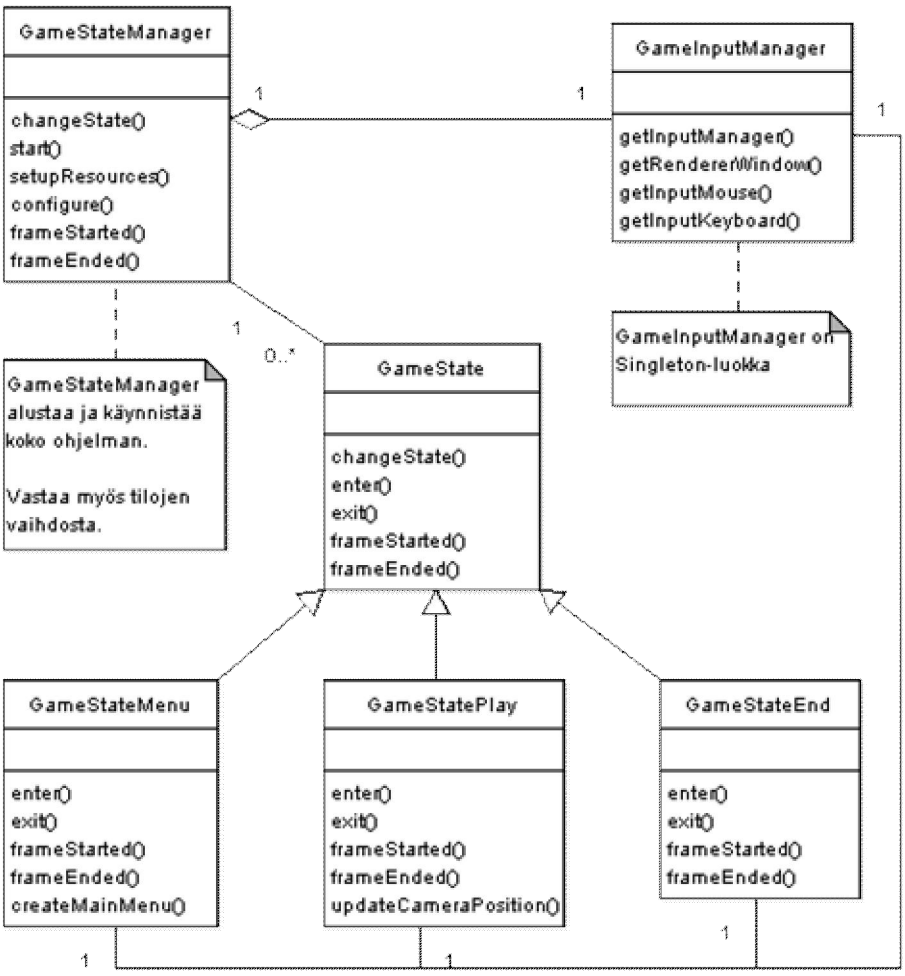
Pelissä on ideana liikuttaa pientä valkoista jänistä pelialueella (kuva 9) ja puolustautua hyökkääviä vihreitä ninjoja vastaan. Jänis käyttää aseenaan taikaa, jolla hyökkäävän ninjan saa katoamaan pelialueelta. Jokaista kuollutta vihollista kohden syntyy kaksi uutta vihollista pelaajan riesaksi. Vihollisilla on aseenaan vain miekat joilla lyödäkseen vihollisen on päästävä aivan pelaajan viereen. Tarkoitus on tappaa mahdollisimman monta vihollista kunnes omat elämäpisteet kuluvat loppuun.

Kuva 9: Pelitilanne

6.1 Pelitilojen hallinta

CrazyBunny-pelin eri tilojen hallinta on toteutettu state-suunnittelumalliin perustuvalla mallilla. Tämä menettely takaa sen, että peliohjelman kulkua on helppo hallita, sekä tarvittaessa laajentaa. Rakenne toimii siten, että kaikki pelin loogiset kokonaisuudet kuten päävalikko ja pelitila toteutetaan omina tilaluokkinaan, jotka ovat toisistaan täysin riippumattomia. Tilaluokista luodaan staattiset ilmentymät heti pelisovelluksen käynnistyttyä.

Tämä tilajärjestelmä toimii pohjana koko sovellukselle, ja se kätkee sisälleen kaikki välttämättömät sekä muutoin tarpeelliset osat joita OGRE-sovelluksissa tulee käyttää. Kuvassa (Kuva 10) on UML-luokkakaavio, josta CrazyBunny-pelin perusrakenne käy ilmi.



Kuva 10: UML-kaavio tilamallista

6.1.1 GameStateManager-luokka

GameStateManager-luokka on koko CrazyBunny-pelin tärkein luokka, sillä kaikki OGRE:n kannalta tärkeimmät ja välttämättömät oliot luodaan tämän luokan start()-funktiossa. Root-olio on käytännössä luotava aina ensin, sillä se toimii pääsyreittinä OGRE-järjestelmään. Tämän jälkeen suoritetaan setupResources()-funktio, joka lataa kaikki käytettävät resurssit. Seuraavaksi suoritetaan configure()-funktio, joka tekee tarvittavat perusasetukset. Jotta pelissä voidaan suorittaa mitään tehtäviä, on lisättävä vähintään yksi kehyskuuntelija (FrameListener). Tämä tehdään mRoot-olion addFrameListener()-funktiolla. Käyttäjältä tulevien syötteiden havaitsemista varten on luotava ilmentymä GameInputManager-luokasta sekä tässä tapauksessa myös haettava ilmentymä OGRE:n InputManager-luokasta. Lopuksi vaihdetaan aktiivi-

nen pelitila changeState()-funktiolla ja aloitetaan hahmonnus mRoot-olion startRendering()-funktiolla.

```
void GameStateManager::start(GameState* state)
{
    this->mRoot = new Root();

    this->setupResources();

    if (!this->configure())
    {
        return;
    }

    this->mRoot->addFrameListener(this);

    new GameInputManager(this->mRenderWindow);
    this->mInputManager = GameInputManager::getSingletonPtr()
    ->getInputManager();

    this->changeState(state);
    this->mRoot->startRendering();
}
```

Ohjelmakoodin ote 1: Start()-funktio

Seuraavassa on kokonaisuudessaan listattuna setupResources()-funktio, joka yksinkertaisesti käy iteroimalla läpi koko resources.cfg-tiedoston ja lisää tähän tiedostoon asetettujen resurssien hakemistopolut järjestelmään.

```
void GameStateManager::setupResources(void)
{
    /// load resource paths from config file
    ConfigFile cf;
    cf.load("resources.cfg");

    /// go through all settings in the file
    ConfigFile::SectionIterator seci = cf.getSectionIterator();

    String secName, typeName, archName;
    while (seci.hasMoreElements())
    {
        secName = seci.peekNextKey();
        ConfigFile::SettingsMultiMap *settings = seci.getNext();
        ConfigFile::SettingsMultiMap::iterator i;
        for (i = settings->begin() ; i != settings->end() ; ++i)
        {
            typeName = i->first;
            archName = i->second;
            ResourceGroupManager::getSingleton().
            addResourceLocation(
            archName, typeName, secName);
        }
    }
}
```

Ohjelmakoodin ote 2: Resurssien lataus

Alla listatun configure()-funktion tehtävä on ladata järjestelmän asetukset ogre.cfg-tiedostosta mRoot-olion restoreConfig()-funktiolla. Mikäli tämä ei onnistu, annetaan käyttäjän itse tehdä asetukset. Seuraava mRoot-olion initialise()-funktio käytännössä aloittaa ohjelman ja palauttaa onnistuessaan RendererWindow-luokan ilmentymän. Lopuksi luetaan aikaisemmin asetetut resurssit järjestelmään. Näistä suurin osa on tehty skriptikielellä, joten OGRE:n on tehtävä tarvittava tulkitseminen.

```
bool GameStateManager::configure(void)
{
    /// load config settings from ogre.cfg
    if (!mRoot->restoreConfig())
    {
        /// if there is no config file, show the configuration dialog
        if (!mRoot->showConfigDialog())
        {
            return false;
        }
    }

    /// Initialise and create a default rendering window
    this->mRenderWindow = mRoot->initialise(true);

    /// Initialises resource groups
    ResourceGroupManager::
    getSingleton().initialiseAllResourceGroups();

    return true;
}
```

Ohjelmakoodin ote 3: Järjestelmän asetukset

Tässä vaiheessa otetaan käyttöön ensimmäinen varsinainen CrazyBunny-pelin tila. Tilaolio lisätään vektoritauluun push_back()-funktiolla, jonka jälkeen suoritetaan tilan enter()-funktio. Kyseinen enter()-funktio toimii tiloissa oletusmuodostimen tapaan.

```
void GameStateManager::changeState(GameState* state)
{

    /// cleanup the current state

    if ( !mStates.empty() )
    {
        mStates.back()->exit();
        mStates.pop_back();
    }

    /// store and init the new state
    mStates.push_back(state);
    mStates.back()->enter();
}
```

Ohjelmakoodin ote 4: Tilan asettaminen

6.1.2 GameInputManager-luokka

GameInputManager-luokan tarkoituksena on pitää jatkuvasti saatavilla CrazyBunny-pelin kaikille tiloille yhteiset palvelut. Luokka on toteutettu Singleton-suunnittelumallilla, jolla varmistetaan se, että luokasta voi olla vain yksi ilmentymä koko järjestelmässä. OGREssa tämä tehdään periyttämällä omalle luokalle Singleton-malli (template).

Oletusmuodostinfunktiossa luodaan tulojen hallinta (InputManager), jonka createInputSystem()-funktiolle annetaan parametrina lista, jonka avaimena on WINDOW ja arvona windowHndStr-olion str()-funktion palauttama m_win. InputManager-luokan funktiolla createInputObject() luodaan tulo-objektit hiirelle ja näppäimistölle. GameInputManager-luokka sisältää myös palautusfunktiot joilla näihin jäsenmuuttujiin päästään käsiksi tilaluokissa (ks. kpl. 6.1.4).

```
GameInputManager::GameInputManager(Ogre::RenderWindow* mRenderWindow)
{
    this->mRenderWindow = mRenderWindow;

    OIS::ParamList pl;
    size_t windowHnd = 0;
    std::ostringstream windowHndStr;

    this->mRenderWindow->getCustomAttribute("WINDOW", &windowHnd);
    windowHndStr << windowHnd;
    pl.insert(std::make_pair(std::string("WINDOW"),
      windowHndStr.str()));

    this->mInputManager = OIS::InputManager::createInputSystem(pl);
    this->mMouse = static_cast<OIS::Mouse*>
      (this->mInputManager->createInputObject(OIS::OISMouse, false));
    this->mKeyboard = static_cast<OIS::Keyboard*>
   (this->mInputManager->createInputObject(OIS::OISKeyboard, false));
}
```

Ohjelmakoodin ote 5: Tulojen luonti

6.1.3 GameState-luokka

Kaikki pelin varsinaiset tilaluokat perivät tämän abstraktin luokan, joka määrittää yhteensä kuusi virtuaalista funktiota, jotka on toteutettava kaikissa tilaluokissa.

```
virtual void enter() = 0;
virtual void exit() = 0;
virtual void pause() = 0;
virtual void resume() = 0;

virtual bool frameStarted(const Ogre::FrameEvent& evt) = 0;
virtual bool frameEnded(const Ogre::FrameEvent& evt) = 0;
```

Ohjelmakoodin ote 6: GameState-luokan abstraktit jäsenfunktiot

Tämän lisäksi tässä luokassa määritetään kolme tilojen vaihtamiseen liittyvää funktiota jotka periytyvät varsinaisille tilaluokille. Näiden funktioiden toteutus on GameStateManager-luokassa (ks. kpl. 6.1.1).

```
void changeState(GameState* state)
{
    GameStateManager::getSingletonPtr()->changeState(state);
}

void pushState(GameState* state)
{
    GameStateManager::getSingletonPtr()->pushState(state);
}

void popState()
{
    GameStateManager::getSingletonPtr()->popState();
}
```

Ohjelmakoodin ote 7: Tilojen vaihtaminen

6.1.4 Varsinaiset tilaluokat

CrazyBunny-pelissä on toteutettu yhteensä kolme tilaluokkaa jotka ovat päävalikko (GameStateMenu-luokka), pelitila (GameStatePlay-luokka) ja pelin loppu (GameStateEnd-luokka). Jokainen tilaluokka toteuttaa GameState-luokan määrittelemää rajapintaa (ks. kpl 6.1.3) sekä käyttävät pelitilan vaihtamiseen perittyä funktioita changeState().

Tietylle tilalle ominaiset operaatiot suoritetaan jokaisella ohjelman kierroksella yleensä frameStarted()-funktiossa, mutta joissakin tapauksissa on järkevää suorittaa tietty operaatio frameEnded()-funktiossa. Molemmat edellä mainitut funktiot kuuluvat OGRE:n hahmonnus-silmukkaan ja ovat määriteltynä OGRE:n FrameListener-luokassa aidosti virtuaalisina.

6.2 Peliohjelman käynnistys

Ohjelman WinMain-funktiossa luodaan ensin ilmentymä GameState-Manager-luokasta, minkä jälkeen suoritetaan kyseisen luokan start()-funktio, jolle annetaan parametrina ilmentymä ensimmäisestä pelitilasta joka tässä tapauksessa on päävälikko (GameStateMenu-luokka). Tilan ilmentymän luonti tapahtuu tilaluokan staattisella getInstance()-funktiolla.

```
INT WINAPI WinMain( HINSTANCE hInst, HINSTANCE, LPSTR strCmdLine, INT
)
{
    using namespace Ogre;
    GameStateManager* stateManager = new GameStateManager();
    try
    {
        stateManager->start(GameStateMenu::getInstance());
    }
    catch( Exception& e )
    {
        MessageBox( NULL, e.getFullDescription().c_str(),
        "An exception has occured!", MB_OK | MB_ICONERROR
        | MB_TASKMODAL);
    }

    return 0;
}
```

Ohjelmakoodin ote 8: Sovelluksen käynnistys

CrazyBunny-pelin pääfunktiossa ei siis tehdä muuta kuin käynnistetään OGRE sekä pelitilojen hallinta ja asetetaan ensimmäinen tila aktiiviseksi.

6.3 Graafisen käyttöliittymän luonti ja hallinta

OGRE ei sisällä omaa järjestelmää käyttöliittymien luontiin joten tätä tehtävää varten on käytettävä ulkopuolista kirjastoa. CEGUI (ks. kpl. 2.5) sopii tähän mainiosti, sillä CEGUI-kirjasto sekä rajapinta toimitetaankin nykyään OGRE:n mukana.

Graaffista käyttöliittymää tarvitaan tässä työssä lähinnä CrazyBunny-pelin päävalikossa ja siinäkin vain muutamien painikkeiden luontiin. CEGUI mahdollistaisi tarvittaessa myös hyvin monimutkaisten käyttöliittymien rakentamisen. Seuraavassa kuvassa (Kuva 11) on pelin päävalikko.

Kuva 11: Pelin päävalikko

6.3.1 Alustukset

CEGUI tukee OGRE:a suoraan siten, että luodaan ilmentymä CEGUI:n luokasta OgreCEGUIRenderer, jolle annetaan parametrina käytössä oleva OGRE:n RendererWindow- sekä SceneManager-olio. Seuraavaksi luodaan ilmentymä System-luokasta ja annetaan tälle parametrina OgreCEGUIRenderer-olio. Kolmantena luodaan pääikkuna johon kaikki loput ikkunat ja painikkeet sisällytetään. Tämä tehdään WindowManager-luokan createWindow()-funktiolla jolle annetaan parametrina käytettävän tyylini nimi sekä tunnus, joka tässä tapauksessa on ”Sheet”.

Käyttöliittymän ulkoasu ladataa XML-tiedostosta SchemeManager-luokan loadScheme()-funktiolla, jolle annetaan parametrina tiedoston nimi, jonka jälkeen OGRE:n resurssienhallinta pitää huolen siitä, että oikea tiedosto tulee ladattua. Seuraava tärkeä operaatio on asettaa System-oliolle aikaisemmin luotu pääikkuna, johon muut ikkunat sisällytetään. Tämä tehdään setGUISheet()-funktiolla, jolle annetaan parametrina pääikkunan ilmentymä.

```
/// Set up GUI system and the logger
this->mOgreGUIRenderer = new CEGUI::OgreCEGUIRenderer(
this->mRenderWindow,Ogre::RENDER_QUEUE_OVERLAY, false, 3000,
this->mSceneMgr);
this->mGUISystem = new CEGUI::System(this->mOgreGUIRenderer);
this->mEditorGuiSheet = CEGUI::WindowManager::getSingleton()
.createWindow((CEGUI::utf8*)"DefaultWindow", (CEGUI::utf8*)"Sheet");

CEGUI::Logger::getSingleton().setLoggingLevel(CEGUI::Informative);

/// Set up GUI scheme and position mouse in the middle of the screen
CEGUI::SchemeManager::getSingleton().loadScheme((CEGUI::utf8*)
"TaharezLookSkin.scheme");
this->mGUISystem->setDefaultMouseCursor((CEGUI::utf8*)"TaharezLook",
(CEGUI::utf8*)"MouseArrow");
CEGUI::MouseCursor::getSingleton().setImage("TaharezLook",
"MouseMoveCursor");
this->mGUISystem->setDefaultFont((CEGUI::utf8*)"BlueHighway-12");
this->mGUISystem->setGUISheet(this->mEditorGuiSheet);
this->mGUISystem->injectMousePosition((float)this->mRenderWindow
->getWidth()/2, (float)this->mRenderWindow->getHeight()/2);
```

Ohjelmakoodin ote 9: Käyttöliittymän alustukset

6.3.2 Painikkeen luonti

Jokaiselle painikkeelle määritetään luontivaiheessa tyyli ja nimi createWindow()-funktion parametreina. Tämän jälkeen luotu painike lisätään käyttöliittymään mEditorGuiSheet-olion addChildWindow()-funktiolla. Painikkeelle asetetaan myös paikka setPosition()-funktiolla, koko setSize()-funktiolla sekä teksti setText()-funktiolla. Paikan ja koon määrittelyyn käytetään CEGUI-kirjaston UVector2-olita.

```
/// Creating a start button here
this->startButton = (CEGUI::PushButton*)
CEGUI::WindowManager::getSingletonPtr()->
createWindow("TaharezLook/Button", "Start");
this->mEditorGuiSheet->addChildWindow(this->startButton);
this->startButton->setPosition(CEGUI::UVector2(cegui_reldim(0.20f),
cegui_reldim( 0.30f)));
this->startButton->setSize(CEGUI::UVector2(CEGUI::UDim(0.15, 0),
CEGUI::UDim(0.05, 0)));
this->startButton->setText("Start");
```

Ohjelmakoodin ote 10: Painikkeen luonti

6.3.3 Käsittelijäfunktion rekisteröinti

Jokaiselle painikkeelle voidaan määrittää subscribeEvent()-funktiolla, mitä tapahtumaa halutaan kuunnella sekä lisäksi käsittelijä tällä tapahtumalle.

Tapahtuman käsittelijä on takaisinkutsu (callback), jonka toteutus on ohjelmoitu CrazyBunny-pelin GameStateMenu-luokaan. Tässä tapauksessa painike kuuntelee EventMouseButtonDown-tapahtumaa joka tarkoittaa sitä, että käsittelijä suoritetaan jos painiketta on hiirellä klikattu.

```
this->startButton->
subscribeEvent(CEGUI::PushButton::EventMouseButtonDown,
CEGUI::Event::Subscriber(&GameStateMenu::handleStartGame, this));
```

Ohjelmakoodin ote 11: Käsittelijän rekisteröinti

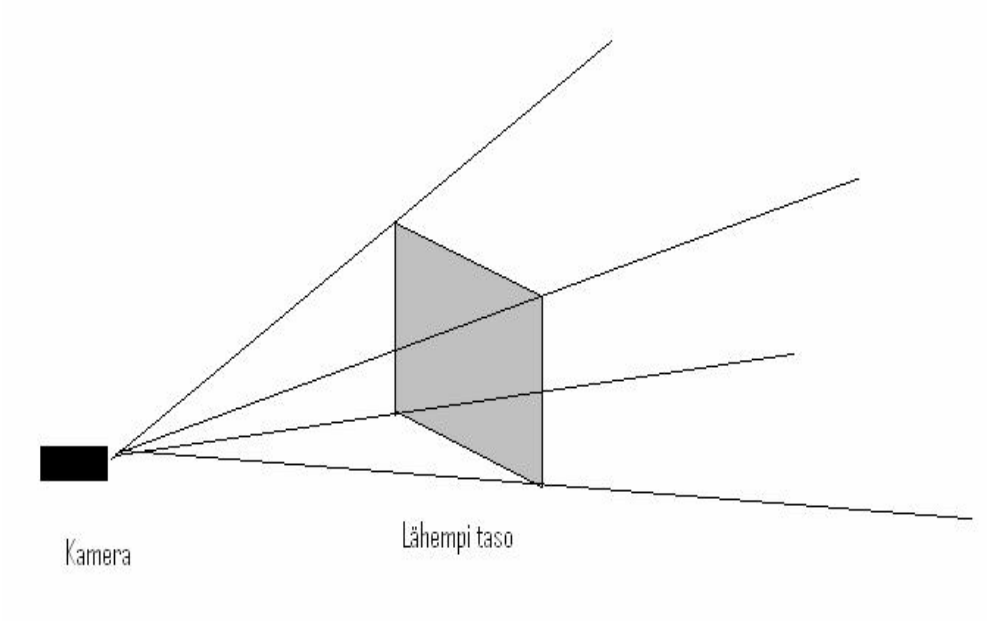
Seuraavassa ohjelmakoodin otteessa on edellisessä ohjelmakoodin otteessa asetetun käsittelijän toteutus. Tämä käytännössä vaihtaa ohjelman tilan päävalikosta pelitilaksi changeState()-funktiolla.

```
bool GameStateMenu::handleStartGame(const CEGUI::EventArgs& e)
{
    // Changing to play state
    this->changeState(GameStatePlay::getInstance());

    return true;
}
```

Ohjelmakoodin ote 12: Tapahtuman käsittelijä

6.4 Kamerajärjestelmä

Kamera on hyvin keskeinen osa nykyajan pelimoottoreissa, sillä pelitilannetta tarkastellaan aina kameran näkökulmasta kuten elokuviakin. Kameran näkymä kolmiulotteiseen maailmaan on loputon neliökantainen pyramidi, jonka kärki on katkaistu (kuva 12). Katkaisusta on se hyöty, että aivan kameran lähellä olevia ilmentymiä ei hahmonneta (render), joten näkymä ei mene koskaan tukkoon. OGRE:n kamera toimii samalla tavalla kuin ympäristösolmut (SceneNode), joten kameraa voidaan pitää näiden erikoistapauksena.



Kuva 12: Kameran näkymä

Kamera luodaan SceneManager-luokan createCamera()-funktiolla jolle annetaan parametrina nimi. Tämä voi olla käytännössä mitä tahansa mutta kahta samannimistä kameraa ei voida luoda. Ensin luodaan ympäristösolmu (SceneNode) jota käytetään kameran paikan laskemiseen. Tämä tehdään luomalla pelaajasolmulle lapsisolmu createChildSceneNode()-funktiolla. Seuraavaksi asetetaan kameran näkökentän katkaiseva lähempi taso setNearClipDistance()-funktiolla, jonka jälkeen kamera siirretään origoon.

```
this->sightNode = this->bunny->getPlayerNode()->
    createChildSceneNode ("SightNode", Vector3(0, 10, 40));
this->mCamera = this->mSceneMgr->createCamera("GameCamera");
this->mCamera->setNearClipDistance(5);
/// Setting camera to origin
this->mCamera->setPosition(Vector3(0,0,0));
```

Ohjelmakoodin ote 13: Kameran asetuksia

Kamerajärjestelmän voi tehdä usealla tavalla. Tässä työssä kameran tulee seurata pelaajan hahmoa hieman yläviistosta. Toteutus on tehty siten, että kameraa varten luodaan ympäristösolmu (SceneNode), johon varsinainen kamera liitetään attachObject()-funktiolla. Tämän jälkeen kamera nostetaan ylös translate()-funktiolla, jolle annetaan parametrina vektori, missä y-komponenttiin asetetaan arvo 5.

Seuraavat kaksi vaihetta ovat erityisen tärkeät kamerajärjestelmän toimivuuden kannalta. Kamera asetetaan katsomaan kohdetta setAutoTracking()-funktiolla, jonka ensimmäinen parametri määrittelee, onko automaattinen kohteen seuraaminen käytössä ja toiseen parametriin asetetaan osoitin ympäristösolmuun (SceneNode), jota halutaan seurata. Ilman seuraavaa setFixedYawAxis()-funktiota kamera alkaa pyöriä käännön yhteydessä, mikäli käytetään automaattista seurantaa (auto tracking). Tämä onkin oikeastaan ainoa tilanne, jossa akselin kiinnitys on tarpeen.

```
this->cameraNode = mSceneMgr->getRootSceneNode()->
createChildSceneNode("CameraNode");
this->cameraNode->attachObject(this->mCamera);
this->cameraNode->translate( Vector3( 0, 5, 0 ));

/// The camera will always look at the camera target
this->cameraNode->setAutoTracking (true, this->bunny->
getPlayerNode());

/// Needed because of auto tracking
this->cameraNode->setFixedYawAxis (true);
```

Ohjelmakoodin ote 14: Automaattinen kohteen seuranta

Lopuksi tarvitaan tapa kameran päivittämiseen, joka tässä työssä on GameStatePlay-luokan updateCameraPosition()-funktio. Funktiossa lasketaan kameralle uusi paikka vähentämällä sightNode-olion paikkatiedosta kameran paikka. Ilman tätä menettelyä kamera seuraisi palaajahahmoa kiinnitetystä pisteestä liikkumatta mukana. Kyseinen funktio suoritetaan frameStarted()-funktiossa jokaisella CrazyBunny-ohjelman kierroksella, jotta kamera pysyy aina ajan tasalla.

```
void GameStatePlay::updateCameraPosition()
{
    /// Handle movement
    Vector3 displacement;

    /// Calculating new camera position
    displacement = (this->sightNode->getWorldPosition() –
        this->cameraNode->getPosition());
    this->cameraNode->translate (displacement);
}
```

Ohjelmakoodin ote 15: Kameran päivitys

6.5 Hiiren ja näppäimistön hallinta

Peleissä keskeinen osa on pelaajan ja pelin vuorovaikutus. Pelaaja käyttää näppäimistöä ja hiirtä hallitakseen pelin kulkua itselleen edullisella tavalla. OGRE:ssa on useita tapoja hallita käyttäjän syötteitä, mutta nykyään yleisin lienee OIS (Object-oriented Input System). Käyttäjän syötteitä voi käytännössä hallinnoida kahdella tavalla, puskuroidusti ja puskuroimattomasti. Ero näiden kahden välillä on se, että puskuroitu järjestelmä toteuttaa kuuntelijarajapintoja, joilla ohjelmalle kerrotaan käyttäjän tekemistä tapahtumista. Tiettyä tapahtumaa vastaava ohjelmakoodi suoritetaan heti. Puskuroimattomassa järjestelmässä jokaisella ohjelman kierroksella voidaan tarkistaa esimerkiksi, onko painike painettu alas ja päätellä tämän perusteella, mitä ohjelman tulee seuraavaksi tehdä.

CrazyBunny-pelissä käyttäjän näppäimistöltä antamat syötteet käsitellään puskuroimattomasti pelitilan handleKeyboardInput()-funktiossa, joka suoritetaan ohjelman jokaisella kierroksella frameStarted()-funktiossa. Puskuroimattomassa järjestelmässä on erityisen tärkeätä suorittaa Keyboard-olion capture()-funktio ennen painikkeiden tarkastuksia. Kyseinen funktio käytännössä tallentaa näppäimistön tilan tällä hetkellä. Tietyn näppäimen tilan voi tämän jälkeen tarkistaa Keyboard-olion isKeyDown()-funktiolla, jolle annetaan parametrina OIS-luokan määrittelemä näppäinkoodi. Itse pelissä näppäimistöä käytetään vain liikkumiseen pelialueen sisällä.

```
bool GameStatePlay::handleKeyboardInput()
{
    this->mKeyboard->capture();

    if (this->mKeyboard->isKeyDown(OIS::KC_ESCAPE))
    {
        this->mExitGame = true;

        return false;
    }

    /// Handling player char movement
    if (this->mKeyboard->isKeyDown (OIS::KC_W))
    {
        this->bunny->getPlayerNode()->
        translate (this->bunny->getPlayerNode()->
        getOrientation ()*Vector3 (0, 0, -100 * this->elapsedTime));
    }
    if (this->mKeyboard->isKeyDown (OIS::KC_S))
    {
        this->bunny->getPlayerNode()->
        translate (this->bunny->getPlayerNode()->
        getOrientation ()*Vector3 (0, 0, 100 * this->elapsedTime));
    }
```

```
    if (this->mKeyboard->isKeyDown (OIS::KC_A))
    {
        //this->playerNode->yaw (Radian (2 * this->elapsedTime));
        this->bunny->getPlayerNode()->
        translate (this->bunny->getPlayerNode()->
        getOrientation () * Vector3 (-70 * this->elapsedTime, 0, 0));
    }
    if (this->mKeyboard->isKeyDown (OIS::KC_D))
    {
        //this->playerNode->yaw (Radian (-2 * this->elapsedTime));
        this->bunny->getPlayerNode()->
        translate (this->bunny->getPlayerNode()->
        getOrientation () * Vector3 (70 * this->elapsedTime, 0, 0));
    }

    return true;
}
```

Ohjelmakoodin ote 16: Näppäimistön syötteiden hallinta

Hiireltä tulevat syötteet käsitellään myös puskuroimattomasti. Tässäkin tapauksessa on tärkeätä suorittaa Mouse-olion capture()-funktio hiiren tilan tallentamiseksi. Tiloihin pääsee käsiksi Mouse-olion getMouseState()-funktiolla, joka palauttaa hiiren tilaolion. Painikkeen tilan tarkistus tehdään hiiren tilaolion buttonDown()-funktiolla, jolle annetaan parametrina OIS-luokan määrittelemä hiiren näppäinkoodi. Hiiren liikuttelu tarkistetaan tilaolion X-komponentin rel-muuttujasta. Muuttujan arvo on 0, mikäli hiirtä ei ole liikutettu; muutoin arvo on liikutuksen suhteellinen määrä. Liikutuksen määrää käytetään laskemaan sopiva kierto pelaajahahmolle.

```
bool GameStatePlay::handleMouseInput()
{
    this->mMouse->capture();

    // Moving player node if mouse is moved
    if (this->mMouse->getMouseState().X.rel != 0)
    {
        this->bunny->getPlayerNode()->
            yaw (Radian ((-(this->mMouse->getMouseState().X.rel / 2)
            * this->elapsedTime)));
    }

    // Checking if right mouse button was pressed
    if (this->mMouse->getMouseState().buttonDown(OIS::MB_Right))
    {
        /// Playing sound
        this->sound->PlaySound(this->soundHeal,
            this->defaultChannel, false);

        /// Calculating current direction
        const Ogre::Quaternion* orientation =
            &this->bunny->getPlayerNode()->getOrientation();
        double x = cos((double)orientation->getYaw().valueRadians());
        double z = sin((double)orientation->getYaw().valueRadians());

        /// Casting the projectile
```

```
            this->agentManager->castProjectile(this->
                bunny->getPlayerNode(), Vector3(x,0,z));
            this->bunny->gainHitPoints(1);

        }

        return true;
    }
```

Ohjelmakoodin ote 17: Hiiren syötteiden hallinta

6.6 Peliympäristön luonti

Yksinkertainen ympäristö syntyy taivaasta, maastosta sekä valosta. Taivas asetetaan SceneManager-luokan funktiolla setSkyDome(), jolle annetaan parametrina käytettävän materiaalin nimi, kaarevuus sekä tekstuurin tiilien määrä. Taivas on käytännössä viisisivuinen laatikko jonka jokaiselle sivulle asetetaan tekstuuri. Tässä tapauksessa tekstuuria väännetään, jotta saadaan aikaan tunne siitä, että taivas olisi oikeasti kaareva. Tekstuurin vääntö aiheuttaa sen, että laatikko ei voi olla pohjaa. Joissain tapauksissa näin on kuitenkin oltava ja silloin tulee käyttää setSkyBox()-funktiota taivaan luontiin, vaikka taivaasta ei tule oikeasti kaarevan näköinen.

Seuraavaksi asetetaan hieman vaaleanpunaista sumua tuomaan tunnelmaa. Tämä onnistuu SceneManager-luokan setFog()-funktiolla jolle annetaan parametrina sumun tyyli sekä väri. Sumua on käytännössä kahta tyyppiä, lineaarista ja eksponentiaalista. Lineaarinen sumu sakenee yhtä paljon jokaista matkayksikköä kohden, kun taas eksponentiaalinen sakenee huomattavasti nopeammin.

Ilman valoja ei peliympäristössä näy mitään joten seuraavaksi asetetaan valo koko ympäristöön. Tämä tehdään SceneManager-luokan setAmbientLight()-funktiolla jolle annetaan parametrina vain haluttu väri. OGRE tukee myös dynaamisia valoja, joita voi liittää muihin ympäristösolmuihin (SceneNode), mutta näitä valoja ei vielä aseteta.

Lopuksi ladataan varsinainen maasto OGREN:n SceneManager-luokan setWorldGeometry()-funktiolla, jolle annetaan parametrina tiedostopolku terrain.cfg-tiedostoon. Maasto generoidaan käytännössä kolmesta normaalista kuvasta, joista ensimmäinen on korkeuskartta (heightmap), toinen on maaston varsinainen tekstuuri ja viimeinen kuvaa maaston yksityiskohtia kuten esimerkiksi pieniä halkeamia. Korkeuskartta (heightmap) on mustavalkoinen

kuva, jossa musta väri tarkoittaa matalaa ja valkoinen korkeaa. Kuvankäsittelyohjelmalla voi siis eri valkoisen sävyjä käyttäen piirtää haluamansa maaston muodot.

```
/// Create a skyDome
this->mSceneMgr->setSkyDome( true, "Examples/CloudySky", 5, 8 );
/// Setting up some pink fog
this->mSceneMgr->setFog(FOG_EXP,
ColourValue(0.9, 0.0, 0.0), 0.0001);
/// Create a light
this->mSceneMgr->setAmbientLight( ColourValue( 8, 8, 1 ) );
/// Setting up world geometry
std::string terrain_cfg("CB_terrain.cfg");
this->mSceneMgr->setWorldGeometry(terrain_cfg);
```

Ohjelmakoodin ote 18: Ympäristön luonti

6.7 Dynaaminen valaistus ja varjot

Varjojen ja valaistuksen käyttö on OGRE:ssa kohtalaisen helppoa ja parhaimmillaan erittäin näyttävää. Kaikki nykyiset pelimoottorit käyttävät jotakin tekniikkaa dynaamisten valojen ja varjojen luomiseen. Tämä on erittäin vaativa osa-alue peliohjelmoinnissa varsinkin hahmonnuksen (rendering) kannalta. OGRE tukee tästä syystä useaa tapaa varjojen luontiin, joista mikään ei ole varsinaisesti täydellinen.

6.7.1 Varjot

Varjoja on käytännössä kolmea tyyppiä, joista tässä työssä käytetty on modulatiivinen sapluunavarjo (SHADOWYPE_STENCIL_MODULATIVE). Tälle varjotyypille on ominaista se, että varjon reunat ovat teräviä eikä muualta tuleva valo heikennä varjon tummuutta. Varjot ovat kuitenkin sopivan aidon näköisiä eikä tämä tekniikka ole turhan raskas näytönohjaimelle. Muita mahdollisia vaihtoehtoja ovat modulatiivinen tekstuurivarjo (SHADOWYPE_STENCIL_MODULATIVE) sekä lisäävä sapluunavarjo (SHADOWTYPE_STENCIL_ADDITIVE), joista viimeisin on raskain ja realistisin varjotyyppi.

Varjot otetaan käyttöön SceneManager-luokan setShadowTechnique()-funktiolla jolle annetaan parametrina haluttu varjotyyppi. Tämän jälkeen voidaan määrittää varjon väri setShadowColour()-funktiolla.

```
/// Lets enable shadows
this->mSceneMgr->
    setShadowTechnique(Ogre::SHADOWTYPE_STENCIL_MODULATIVE);
this->mSceneMgr->
    setShadowColour(Ogre::ColourValue(0.9f,0.9f,0.9f));
```

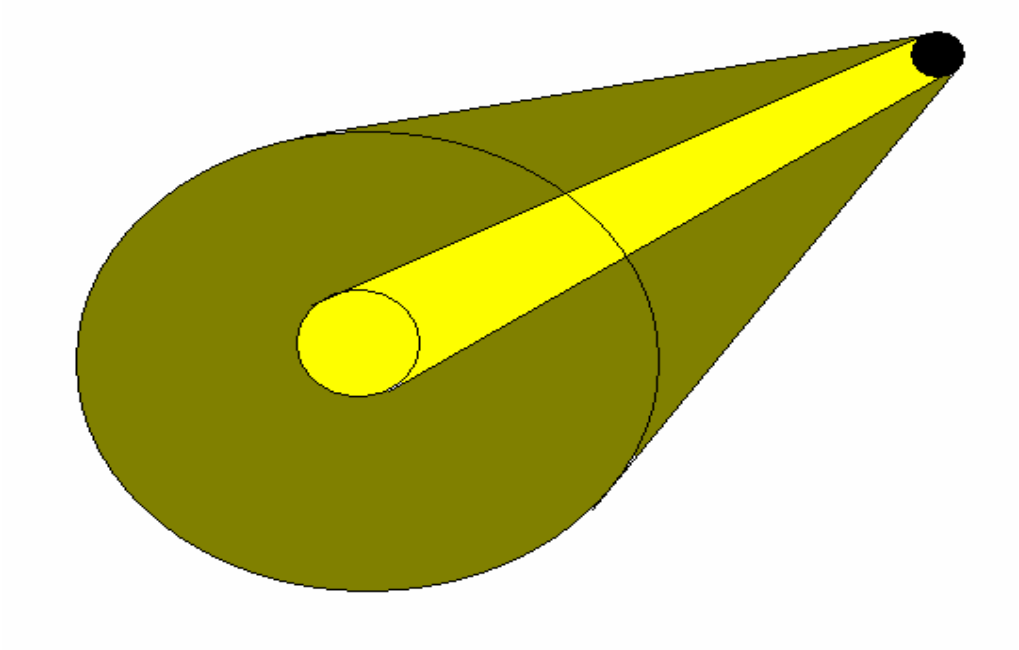
Ohjelmakoodin ote 19: Varjojen käyttöönotto

Jotta tämän jälkeen ruudulla näkyvä ilmentymä (Entity) tekisi varjon, on käytettävä Entity-luokan setCastShadows()-funktiota, jolle annetaan parametrina totuusarvo.

6.7.2 Valaistus

OGRE:ssa on tällä hetkellä mahdollista luoda kolmea valotyyppiä, jotka ovat pistevalo (LT_POINT), valokeila (LT_SPOTLIGHT) ja suunnattuvalo (LT_DIRECTIONAL). Pistevalo valaisee pisteestä joka suuntaan kuten aurinko. Suunnattu valo simuloi, sitä miten valo osuu ympäristöön tietystä suunnasta ja valokeila toimii kuten taskulampun valokeila. CrazyBunny-pelissä käytetään yhtä kohtalaisen korkealle asetettua valokeilaa varjojen luomiseen. Light-luokka perii MovableObject-luokan, joten valot toimivat samaan tapaan kuin ympäristösolmut (SceneNode) ja kamera.

Uusi valo luodaan SceneManager-luokan createLight()-funktiolla, jolle annetaan parametrina valon nimi. Tyyppi asetetaan Light-luokan setType()-funktiolla, jolle annetaan parametrina valon tyyppi. Heikennys (attenuation) asetetaan Light-luokan setAttenuation()-funktiolla, jolle annetaan parametrina etäisyys, jonka jälkeen valo ei enää vaikuta. Loput parametrit vaikuttavat siihen kuinka heikennys lasketaan. Valon suunta asetetaan Light-luokan setDirection()-funktiolla jolle annetaan parametrina suuntavektori. Tässä tapauksessa on käytetty NEGATIVE_UNIT_Y-vektoria, joka tarkoittaa sitä, että valo on suunnattu suoraan alas. Viimeiseksi asetetaan valon kulma eli laajuus. Ensimmäinen parametri on kirkkaan sisemmän valokeilan kulma ja toinen parametri on heikemmän uloimman valokeilan kulma (kuva 13). Sisempi valokeila toimii kuitenkin vain Direct3D-hahmonnuksessa.



Kuva 13: Valokeila

```
/// Adding a spotlight for shadow casting
Light* light = this->mSceneMgr->createLight("Light1");
light->setType(Light::LT_SPOTLIGHT);
light->setDiffuseColour(ColourValue(0.25f,0.25f,0.0f));
light->setSpecularColour(ColourValue(0.25f,0.25f,0.0f));
light->setAttenuation(8000,1,0.0005,0);
light->setDirection(Vector3::NEGATIVE_UNIT_Y);
light->setSpotlightRange(Ogre::Degree(60), Ogre::Degree(70));
```

Ohjelmakoodin ote 20: Valokeilan luonti

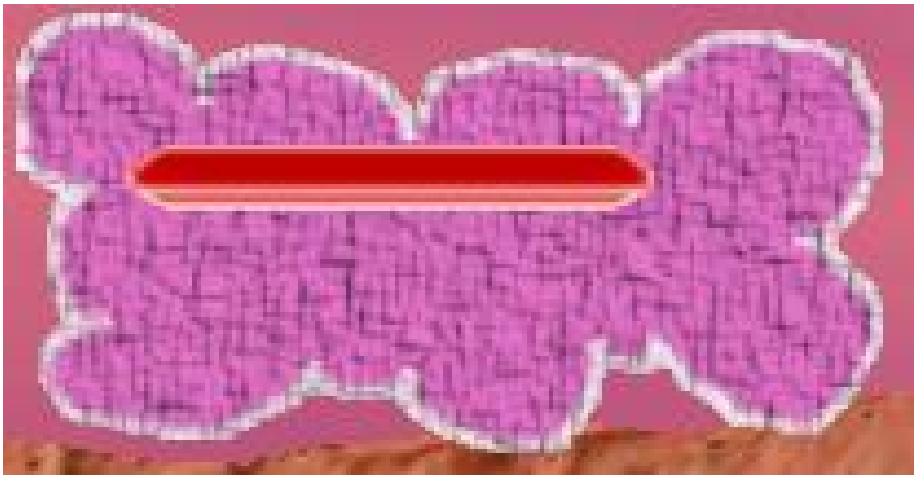
Valoa ei ole pakko liittää ympäristösolmuun (SceneNode) mutta tässä työssä valo liitetään ympäristösolmuun, johon liitetään lisäksi myös ilmentymä (Entity). Valo on käytännössä niin korkealla, että kamera ei missään vaiheessa näe ilmentymää. Tuntuu kuitenkin loogiselta ajatella valokeilaa aurinkona ja lisätä sille ulkomuoto. Varsinkin pelin kehitysvaiheessa on hyödyllistä pystyä helposti paikantamaan valot.

```
/// Adding a scene node for the light
SceneNode* lightNode = mSceneMgr->getRootSceneNode()->
    createChildSceneNode("LightNode");
lightNode->attachObject(light);
lightNode->setPosition(0,300,0);
Entity* lightSphere = this->mSceneMgr->
    createEntity("LightSphere","sphere.mesh");

/// Materializing the light
lightSphere->setMaterialName("Examples/Hilite/Yellow");
lightNode->attachObject(lightSphere);
lightNode->setScale(0.05f,0.05f,0.05f);
```

Ohjelmakoodin ote 21: Valon lisääminen ympäristöön

6.8 Kuva-alustat (Overlay)

Kuva-alusta on ruudulla näkyvä kaksiulotteinen elementti, jota voidaan käyttää esimerkiksi paneelina, joka näyttää pelaajalle tietoa pelin kulusta. Kuva-alustoja ei suoraan voi käyttää varsinaisena käyttöliittymänä sillä tarvittavaa tapahtumankäsittelyä ei ole. Tässä työssä Kuva-alustaa on käytetty ruudun ylälaidassa näyttämään pelaajalle jäljellä olevien elämäpisteiden määrä (kuva 14).



Kuva 14: Kuva-alusta

Kuvassa näkyvä kuva-alusta luodaan skriptikielellä joka lataa kuvankäsittelyohjelmassa piirretyn taustakuvan sekä punaisen vaakapalkin. Seuraavassa ohjelmakoodin otteessa on listattuna kuva-alustan määrittely, joka koostuu kahdesta osasta. Ensimmäinen on container-määritys, joka käytännössä on kuva-alustan tausta. Taustalle määritetään koko, paikka, reuna sekä asetetaan mittayksikkö pikseleiksi. Toinen on element-määritys, jolla taustan päälle piirretään punainen palkki

```
CrazyBunny/Overlay
{
    zorder 500
    container BorderPanel(CrazyBunny/Panel)
    {
        metrics_mode pixels
        vert_align top
```

```
        left 3
        top 3
        width 200
        height 100
        material CrazyBunny/Panel
        border_size 0 0 0 0

        element BorderPanel(CrazyBunny/Health)
        {
            metrics_mode pixels
            left 25
            top 30
            width 120
            height 15
            material CrazyBunny/Health
            border_size 0 0 0 0
        }
    }
}
```

Ohjelmakoodin ote 22: Kuva-alusta

Kuva-alusta ladataan ohjelmassa OGRE:n OverlayManager-luokan getByName()-funktiolla, jolle annetaan parametrina ladattavan kuva-alustan nimi. Resurssien hallinta löytää automaattisesti oikean tiedoston ladatuista resurssiryhmistä (ks. 6.1.1).

```
void GameStatePlay::createOverlays()
{
    this->overlay = OverlayManager::getSingleton()
       .getByName("CrazyBunny/Overlay");
    this->overlay->show();
}
```

Ohjelmakoodin ote 23: Kuva-alustan luonti

Kun kuva-alusta on luotu, pitää sitä päivittää jokaisella ohjelman kierroksella frameStarted()-funktiossa siten, että ensin haetaan haluttu elementti OverlayManager-luokan getOverlayElement()-funktiolla, jolle annetaan parametrina elementin nimi. Tämän jälkeen asetetaan elementin pituus vastaamaan pelaajahahmon jäljellä olevia elämäpisteitä.

```
void GameStatePlay::updateOverlays()
{
    try
    {
        OverlayElement* health = OverlayManager::getSingleton()
            .getOverlayElement("CrazyBunny/Health");
        health->setWidth(this->bunny->getHitPoints());

    }
    catch(...)
    {
```

```
        // ignore
    }
}
```

Ohjelmakoodin ote 24: Kuva-alustan päivitys

6.9 Liikkuvat kappaleet

Tämän työn lopputuloksena syntyvässä CrazyBunny-pelissä on kolmea erityyppistä liikkuvaa kappaletta, joista ensimmäinen on pelaajahahmo, toinen on pelaajahahmon ammus ja kolmas on vihollinen. Pelin AIAgentManager-luokkaa käytetään ammusten ja vihollisten luomiseen sekä hallinnointiin.

Vihollisten ja ammusten hallinnointiin käytetään samaa periaatetta kuin alla esitetyssä spawnAgent()-funktiossa. Kummankin kappaletyypin ilmentymiä pidetään vektoritaulussa, jotta uusien kappaleiden lisääminen ja käsittely olisi mahdollisimman helppoa. Aina ennen uuden kappaleen luontia tarkistetaan onko vektoritaulussa kuolleita kappaleita. Mikäli näin on, otetaan kuollut kappale uudestaan käyttöön, eikä uutta kappaletta jouduta luomaan. Tämä parantaa pelin suorituskykyä. Kappaleet voidaan tyhjentää ja alustaa uudestaan reset()-funktiolla, jonka jälkeen ne toimivat aivan kuten täysin uusi kappale.

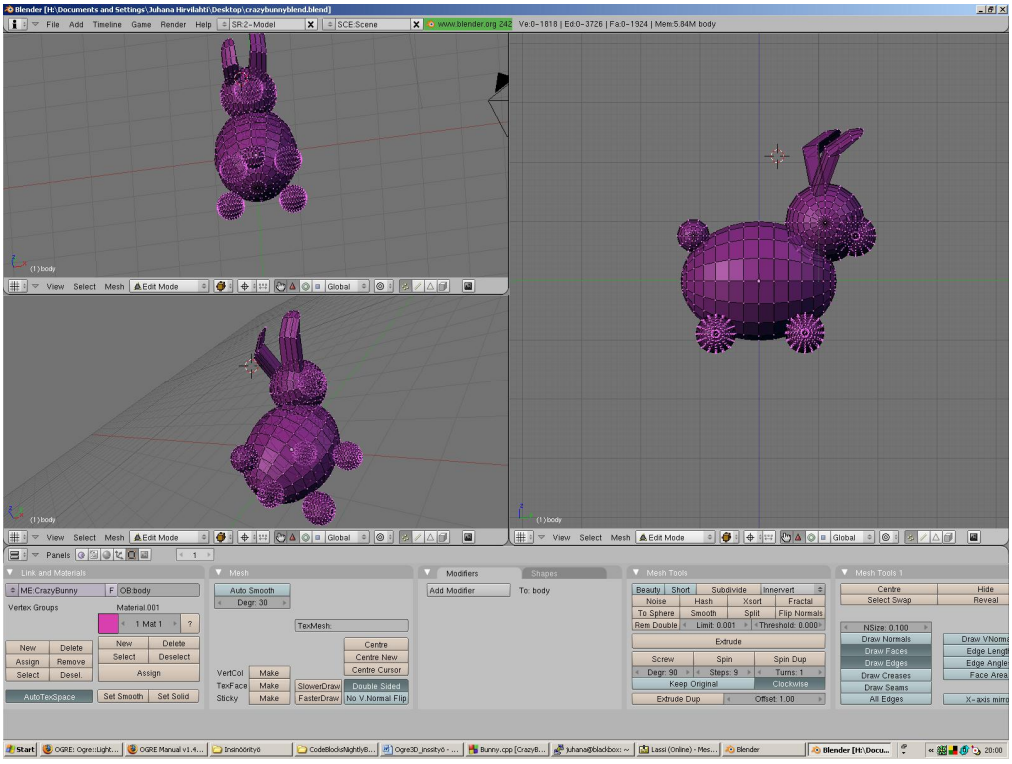
```
void AIAgentManager::spawnAgent(Ogre::SceneNode* node)
{
    srand(time(NULL));
    int num = rand() % 4;
    bool respawn = false;

    for (int i = 0; i < this->agentsV.size(); i++)
    {
        if (this->agentsV[i]->isAlive() == false)
        {
            this->agentsV[i]->reset(this->spawnPoints[num]);
            this->agentsV.back()->setTarget(this->targetNode);

            respawn = true;
        }
    }

    if (this->agentsV.size() < this->agentsV.max_size() &&
        respawn == false)
    {
        this->agentsV.push_back(new AIAgent(this->mSceneMgr,
            this->spawnPoints[num], this->guid));
        this->agentsV.back()->setTarget(this->targetNode);
        this->guid++;
    }
}
```

Ohjelmakoodin ote 25: Vihollisten luonti pelin aikana

6.9.1 Pelaajahahmo

Pelaajahahmoa varten on luotu oma Bunny-luokka, joka oletusmuodostimessa luodaan hahmoa varten ympäristösolmun (SceneNode) ja ilmentymän (Entity). Tämän lisäksi luokassa on hitPoints-jäsenmuuttuja, jonka arvon tulee olla yli 0, jotta peli voi jatkua. Mikäli vihollinen törmää palaajahahmoon, vähennetään elämäpisteitä Bunny-luokan looseHitPoints()-funktiolla.

Pelaajahahmo on yksinkertainen Blender:llä mallinnettu jänis (kuva 15), joka koostuu lähinnä palloista. Mallin korvat on erikseen venytetty jäniksen päänä toimivasta pallosta. Jotta Blenderillä mallinnetun hahmon voi ottaa käyttöön OGRE:ssa, on käytettävä OgreXMLConverter-ohjelmalaajennusta. Tämä luo OGRE:n ymmärtämän mallin ja materiaalin jotka voi suoraan ladata pelissä.



Kuva 15: Hahmon mallinnus

6.9.2 Ammukset

CrazyBunny-peliin on tässä työssä luotu ammuksia varten oma Projectile-luokka, jonka tärkein ominaisuus on go()-funktio. Tämä funktio tarkistaa, ensin onko ammus ”elossa” eli onko timeToLive-muuttujan arvo yli nolla ja tämän jälkeen siirtää ammusta menosuuntaan. Lopuksi vähennetään ammuksen timeToLive-muuttujan arvoa.

```
void Projectile::go(Ogre::Real elapsedTime)
{
    if (this->isAlive())
    {
        Ogre::Vector3 projectilePosition = this->
            projectileNode->getPosition();
        projectilePosition.z -=  this->proDirection.x * 400
            * elapsedTime;
        projectilePosition.x -=  this->proDirection.z * 400
            * elapsedTime;
        this->projectileNode->setPosition(projectilePosition);

        this->timeToLive -= 200 * elapsedTime;
    }
}
```

Ohjelmakoodin ote 26: Ammuksen liikutus

Ammus on normaali ympäristösolmu (ScceneNode), johon on ilmentymän (Entity) sijaan liitetty visuaalinen tehoste (particle effect). Tehoste ladataan SceneManager-luokan createParticleSystem()-funktiolla ja liitetään ympäristösolmuun attach()-funktiolla. Näin saadaan aikaan hieno ja värikäs visuaalinen tehoste (kuva 16).

Kuva 16: Visuaalinen tehoste

Kuten materiaalit ja kuva-alustat (ks. kpl. 6.8) luodaan visuaaliset tehosteetkin skriptikielellä seuraavan ohjelmakoodin otteen mukaisesti.

```
CrazyBunny/Fountain
{
    material        Examples/Flare2
    particle_width  20
    particle_height 40
    cull_each       false
    quota           10000
    billboard_type  oriented_self

    // Area emitter
    emitter Point
    {
        angle           30
        emission_rate   80
        time_to_live    1
        direction       0 1 0
        velocity_min    50
        velocity_max    80
        colour_range_start  0 0 0
        colour_range_end    1 0 1
    }

    // Gravity
    affector LinearForce
    {
        force_vector      0 -120 0
        force_application add
    }
```

```
    // Fader
    affector ColourFader
    {
        red -0.25
        green -0.25
        blue -0.25
    }
}
```

Ohjelmakoodin ote 27: Visuaalinen tehoste

6.9.3 Viholliset

Vihollisia varten on peliin luotu oma AIAgent-luokka, jonka oletusmuodostimessa luodaan ympäristösolmu (SceneNode) ja ilmentymä (Entity). Tärkein ominaisuus vihollisille on pelaajan etsiminen pelialueelta. Tämä on toteutettu AIAgent-luokan findPlayer()-funktiossa. Kyse on todella yksinkertaisesta algoritmista, jossa vihollisen koordinaattien arvoja lisätään tai vähennetään, kunnes kohteen ja vihollisen koordinaattien erotus on nolla. Erotuksen ollessa nolla ovat pelaajahahmo ja vihollinen samassa avaruuden pisteessä. Vihollinen käännetään myös katsomaan pelaajahahmoa kohti SceneNode-luokan lookAt()-funktiolla, jolle annetaan parametrina pelaajahahmon paikkatiedot.

```
void AIAgent::findPlayer(Ogre::Real elapsedTime)
{
    if (this->isAlive())
    {
        Ogre::Vector3 targetVector = this->targetNode->getPosition();
        Ogre::Vector3 agentVector = this->agentNode->getPosition();

        if (targetVector.x - agentVector.x > 0)
        {
            agentVector.x += (50 * elapsedTime);
        }

        if (targetVector.z - agentVector.z > 0)
        {
            agentVector.z += (50 * elapsedTime);
        }

        if (targetVector.x - agentVector.x < 0)
        {
            agentVector.x -= (50 * elapsedTime);
        }

        if (targetVector.z - agentVector.z < 0)
        {
            agentVector.z -= (50 * elapsedTime);
        }
```

```
        /// Setting position ans turning towards target node
        this->agentNode->setPosition(agentVector);
        this->agentNode->lookAt(this->targetNode->getPosition(),
            Ogre::Node::TS_WORLD);
    }
}
```

Ohjelmakoodin ote 28: Pelaajahahmon etsintä

Vihollisten ilmentymänä (Entity) käytetään valmista mallia, joka sisältää useita animaatioita. Valmiiden animaatioiden käyttäminen on OGRE:ssa helppoa. Tätä varten on pelisovellukseen luotu kaksi funktiota. Ensimmäisessä setAnimationState()-funktiossa haetaan käytettävä animaatio Entity-luokan getAnimationState()-funktiolla, jolle annetaan parametrina halutun animaation nimi. Tämän jälkeen animaatio asetetaan silmukkaan setLoop()-funktiolla. Lopuksi animaatio pitää asettaa aktiiviseksi setEnabled()-funktiolla jolle annetaan parametrina totuusarvo. Seuraavan updateAnimationTime()-funktion tehtävä on päivittää animaation ajastus, jotta suoritus tapahtuu kaikilla tietokoneilla yhtä nopeasti. Tämä tehdään AnimationState-luokan addTime()-funktiolla, jolle annetaan parametrina edelliseen ohjelmakierrokseen kulunut aika.

```
void AIAgent::setAnimationState()
{
    this->agentAnimState = this->
        agentEntity->getAnimationState("Walk");
    this->agentAnimState->setLoop(true);
    this->agentAnimState->setEnabled(true);
}

void AIAgent::updateAnimationTime(Ogre::Real time)
{
    this->agentAnimState->addTime(time);
}
```

Ohjelmakoodin ote 29: Animaatio

6.10 Törmäyksen tarkistus

Törmäyksillä tarkoitetaan sitä hetkeä, jolloin kaksi avaruuden kappaletta koskevat toisiaan eli ovat osittain tai kokonaan samassa pisteessä toisen kappaleen kanssa. Tämä on tärkeätä, kun halutaan tietää, onko ammus osunut viholliseen tai onko vihollinen osunut pelaajahahmoon.

Törmäyksiä voi tarkistaa todella tarkastikin mutta tässä työssä käytetään yksinkertaisinta tapaa joka perustuu kaikkien kappaleiden ympärillä olevaan rajalaatikkoon (boungind box). Rajalaatikko haetaan Entity-luokan getWorldBoundingBox()-funktiolla ja rajalaatikon getMaximum()-funktiolla voidaan hakea vektori, joka määrittää laatikon rajat.

Ensin haetaan rajalaatikko (boungind box) pelaajahahmolle, jotta arvoja voidaan verrata vihollisten kanssa. Kaikki viholliset käydään läpi for-silmukassa ja tarkistetaan, onko pelaaja vihollisen rajalaatikon alueella. Mikäli näin on, vähennetään pelaajan elämäpisteitä. Sisäkkäisessä for-silmukassa tehdään tämä käytännössä sama tarkistus ammuksille ja vihollisille. Vihollinen menettää elämäpisteitä mikäli, ammus on vihollisen rajojen sisäpuolella. Samalla ammus poistetaan pelistä, jotta yhdellä ammuksella ei voi osua useampaa kuin yhtä vihollista.

```
void AIAgentManager::checkCollisions(Bunny* bunny)
{
    Ogre::Entity* playerEntity = bunny->getPlayerEntity();
    Ogre::AxisAlignedBox aabbPlayer = playerEntity->
        getWorldBoundingBox();

    for (int i = 0; i < this->agentsV.size(); i++)
    {
        Ogre::Entity* agentEntity = this->agentsV[i]->
            getAgentEntity();
        Ogre::AxisAlignedBox aabbAgent = agentEntity->
            getWorldBoundingBox();

        if (   aabbAgent.getMaximum().x > aabbPlayer.getMinimum().x
            && aabbAgent.getMaximum().z > aabbPlayer.getMinimum().z
            && aabbAgent.getMinimum().x < aabbPlayer.getMaximum().x
            && aabbAgent.getMinimum().z < aabbPlayer.getMaximum().z)
        {
           bunny->looseHitPoints(1);
        }

        for (int k = 0; k < this->projectiles.size(); k++)
        {
            if (this->agentsV[i]->isAlive() && this->projectiles[k]->
                isAlive())
            {
                Ogre::SceneNode* projectileNode = this->
                    projectiles[k]->getProjectileNode();

                if (   aabbAgent.getMaximum().x > projectileNode->
                    getPosition().x
                    && aabbAgent.getMaximum().z > projectileNode->
                        getPosition().z
                    && aabbAgent.getMinimum().x < projectileNode->
                        getPosition().x
                    && aabbAgent.getMinimum().z < projectileNode->
                        getPosition().z)
                {
                    this->projectiles[k]->remove();
```

```
                    this->agentsV[i]->looseHitPoints(1);
                    this->spawnAgent(bunny->getPlayerNode());
                    this->spawnAgent(bunny->getPlayerNode());
                }
            }
        }
    }
}
```

Ohjelmakoodin ote 30: Törmäyksen tarkistus

6.11 Äänijärjestelmä

Jokainen nykyaikainen peli tarvitsee äänijärjestelmän, joka toimii kolmiulotteisissa tiloissa. FMOD tarjoaa tähän erittäin helppokäyttöisen ratkaisun, jota käytetään suosituissa kaupallisissa peleissä kuten World Of Warcraftissa ja BioShockissa. Tässä työssä FMOD-kirjastoa käytetään lähinnä musiikin soittamiseen sekä tämän lisäksi muutamiin äänitehosteisiin. FMOD-kirjastolle on peliin rakennettu kääreluokka SoundManager, jota käytetään äänijärjestelmän alustukseen, äänten luontiin ja soittamiseen.

Helpoin tapa alustaa äänijärjestelmä on kutsua FMOD_System_Init()-funktiota joka asettaa äänikortin oletusasetuksilla. Tätä ennen on kuitenkin luotava varsinainen järjestelmä FMOD_System_Create()-funktiolla. FMOD_System_Init()-funktio ottaa neljä parametria, joista ensimmäinen on FMOD_System_Create()-funktolla luotu system-olio; toinen on virtuaalisten äänien määrä. Virtuaalisten äänien määrä tarkoittaa sitä kuinka monta ääntä järjestelmässä voidaan soittaa samanaikaisesti. Normaalisti äänikortti tukee vain 32-64 samanaikaista ääntä kortista riippuen. Seuraava parametri on lippu jolla järjestelmän voi alustaa eri tavoilla mutta tässä tapauksessa käytetään normaalia alustusta. Viimeinen parametri on ajurille lähetettävää erikoistietoa varten, mutta tämä on ohitettu asettamalla arvo nollaksi.

```
void SoundManager::InitializeSound()
{
    /// Create the main system object.
    this->result = FMOD_System_Create(&this->system);
    this->CheckError(this->result);

    /// Initialize FMOD.
    this->result = FMOD_System_Init(this->system,
        100, FMOD_INIT_NORMAL, 0);
    this->CheckError(this->result);
}
```

Ohjelmakoodin ote 31: Äänijärjestelmän alustus

Uusi ääni pitää ennen soittamista luoda FMOD_System_CreateSound()-funktiolla, jolle annetaan parametrina system-olio, äänitiedoston nimi, avaustapalippu, lisäinformaatiot, ja osoitin FMOD_SOUND tyyppiseen luokkaan. Avaustapa tässä työssä on FMOD_SOFTWARE, joka käytännössä tarkoittaa sitä, että ääni käsitellään ohjelmallisesti. Oletus on FMOD_HARDWARE mutta tämä ei tue äänen kaikkia käsittelyyn liittyviä tapoja. Lisäinformaatiot ohitetaan asettamalla parametrin arvoksi nolla. Tällä parametrilla voi vaikuttaa esimerkiksi siihen, kuinka äänitiedosto ladataan.

```
FMOD_SOUND* SoundManager::CreateSound(const char* file)
{
    FMOD_SOUND* sound;

    this->result = FMOD_System_CreateSound(this->system,
        file, FMOD_SOFTWARE, 0, &sound);
    this->CheckError(this->result);

    return sound;
}
```

Ohjelmakoodin ote 32: Äänen luonti

Ääni soitetaan FMOD_Sound_PlaySound()-funktiolla mutta tätä ennen voi halutessaan asettaa äänen soimaan kerran tai silmukassa. Tämä tehdään FMOD_Sound_SetMode()-funktiolla, jolle annetaan parametrina osoitin käsiteltävään ääneen sekä tila, joka tässä tapauksessa on FMOD_LOOP_NORMAL. FMOD_Sound_PlaySound()-funktio toimii siten, että parametrina annetaan system-olio, kanavaindeksi, ääni, totuusarvo onko ääni pysäytettynä sekä käytettävä kanava. Kanavaan voi myös asettaa arvon nolla, mikäli ei halua säilyttää kanavaosoitinta. Osoitin kannattaa säilyttää, jos esimerkiksi silmukassa soitettavan musiikin haluaa pysäyttää. Kanavaindeksi on käytännössä aina FMOD_CHANNEL_FREE, joka tarkoittaa sitä, että FMOD:n kanavien hallinta automaattisesti valitsee käyttämättömän kanavan äänen soittamiseen.

```
void SoundManager::PlaySound(FMOD_SOUND* sound,
    FMOD_CHANNEL* channel, bool loop)
{
    if (loop)
    {
        /// Setting loop mode
        this->result = FMOD_Sound_SetMode(sound, FMOD_LOOP_NORMAL);
        this->CheckError(this->result);
```

```
    }

    /// Playing the sound
    this->result = FMOD_System_PlaySound(this->system,
        FMOD_CHANNEL_FREE, sound, false, &channel);
    this->CheckError(this->result);
}
```

Ohjelmakoodin ote 33: Äänen soittaminen

CrazyBunny-pelissä yksittäisen äänen soittaminen tehdään siten, että ensin luodaan ilmentymä pelin SoundManager-luokasta, jonka jälkeen kutsutaan kyseisen luokan InitializeSound()-funktiota. Seuraavaksi luodaan uusi ääni CreateSound()-funktiolla ja tallennataan osoitin musicHandler-muuttujaan. Ääni soitetaan PlaySound()-funktiolla, jolle annetaan parametrina osoitin ääneen, osoitin käytettävään kanavaan sekä totuusarvo, joka määrittää, soitetaanko ääni silmukassa vai ei.

```
FMOD_SOUND* musicHandler;
FMOD_CHANNEL* musicChannel;
SoundManager* sound = new SoundManager();
sound->InitializeSound();
musicHandler = this->sound->CreateSound("music.wav");
sound->PlaySound(musicHandler, musicChannel, true);
```

Ohjelmakoodin ote 34: Äänen käsittely

7 YHTEENVETO

Työn tavoitteena oli tutkia ilmaisten työkalujen tarjoamaa mahdollisuutta tietokonepelien tekoon. Tämä tavoite täyttyi odotusten mukaisesti vaikka avoimeen lähdekoodiin perustuvien työkalujen dokumentointi on paikoitellen todella puutteellista. OGRE tuntuu olevan yksi parhaimpia ilmaisia grafiikkamoottoreita mutta pelinkehittäjälle OGRE:n käyttäminen voi välillä olla turhauttavaa, sillä OGRE ei ole pelimoottori. Järjestelmän joustavuus toki luo rajattomasti mahdollisuuksia ja tilaa tehdä asiat omalla tavalla. Usein joutuu käytännössä kuitenkin käyttämään ulkopuolisia kirjastoja tiettyihin tehtäviin. Tämä tuntuu heikentävän kokonaisuuden joustavuutta.

Työn tuloksena syntyi CrazyBunny niminen, yksinkertainen peli, jossa on pyritty toteuttamaan mahdollisimman monta peleille tyypillistä perusominaisuutta. Käyttöliittymien tekoon on käytetty CEGUI-kirjastoa, joka alun ongelmien

jälkeen tuntuu toimivan moitteetta. Äänet peliin on tehty FMOD-kirjastolla, jonka C++-rajapintaa ei voitu käyttää, sillä se ei toimi MinGW:n C++-kääntäjän kanssa. C-rajapinta kuitenkin toimii ja on todella helppo käyttää. FMOD-kirjasto on erittäin hyvä vaihtoehto peliäänien toteuttamiseen, tosin se ei ole ilmainen kaupallisissa sovelluksissa. Tämän lisäksi kokonaisuuden toteuttamiseen on tarvittu Blender-mallinnusohjelmaa ja Audacity-äänieditoria. Blender on todella hankala ohjelma aloittelijalle mutta ohjeita on onneksi runsaasti saatavilla. Tuntien harjoittelun jälkeen mallintaminen alkaa sujua ja ohjelman tehokkuus käy ilmi.

Kehitysympäristönä käytetty Code::Block on todella hyvän tuntuinen sovellus ja helppo käyttää. Yksi syy tähän lienee se, että Code::Blocks on puhtaasti C++-kehitysympäristö eikä muita ohjelmointikieliä ole tuettu. Code::Blocks vaikuttaa olevan suosittu juuri peliohjelmoijien keskuudessa, sillä C++ on tällä hetkellä paras ohjelmointikieli digitaalisen viihteen tuottamiseen.

VIITELUETTELO

[1] Code::Blocks Studio - The open source, cross platform Free C++ IDE [verkkodokumentti]. [viitattu 10.3.2007]. Saatavissa http://www.codeblocks.org/.

[2] Blender – The free open source 3D content creation suite [verkkodokumentti]. [viitattu 7.5.2007]. Saatavissa http://www.blender.org.

[3] OGRE 3D – Open source graphics engine [verkkodokumentti]. [viitattu 7.5.2007]. Saatavissa http://www.ogre3d.org.

[4] MinGW – Minimalist GNU for Windows [verkkodokumentti]. [viitattu 7.5.2007]. Saatavissa http://www.mingw.org/.

[5] CEGUI – Crazy Eddie's GUI System [verkkodokumentti]. [viitattu 7.5.2007]. Saatavissa http://www.cegui.org.uk/wiki/index.php/Main_Page.

[6] FMOD – Music & soundeffect system [verkkodokumentti]. [viitattu 7.5.2007]. Saatavissa http://www.fmod.org/wiki/index.php5?title=Main_Page.